Encoding logical theories of programs

UNIVERSITÀ DEGLI STUDI DI PISA
DIPARTIMENTO DI INFORMATICA

DOTTORATO DI RICERCA IN INFORMATICA
Università di Pisa–Genova–Udine

Ph.D. Thesis: TD-7/97

Encoding Logical Theories of Programs

Marino Miculan

Abstract. Nowadays, in many critical situations (such as on-board software), it is mandatory to *certify* programs and systems, that is, to prove formally that they meet their specifications. To this end, many logics and formal systems have been proposed for reasoning rigorously on properties of programs and systems. Their usage on non-trivial cases, however, is often cumbersome and error-prone; hence, a *computerized proof assistant* is required. This thesis is a contribution to the field of *computer-aided formal reasoning*.

In recent years, *Logical Frameworks* (LF's) have been proposed as general metalanguages for the description (*encoding*) of formal systems. LF's streamline the implementation of proof systems on a machine; moreover, they allow for conceptual clarification of the object logics. The encoding methodology of LF's (based on the *judgement as types*, *proofs as λ-terms* paradigm) has been successfully applied to many logics; however, the encoding of the many peculiarities presented by formal systems for program logics is problematic.

In this thesis we propose a general methodology for adequately encoding formal systems for reasoning on programs. We consider *Structured and Natural Operational Semantics*, *Modal Logics*, *Dynamic Logics*, and the *μ-calculus*. Each of these systems presents distinctive problematic features; in each case, we propose, discuss and prove correct, alternative solutions. In many cases, we introduce new presentations of these systems, in Natural Deduction style, which are suggested by the metalogical analysis induced by the methodology. At the metalogical level, we generalize and combine the concept of *consequence relation* by Avron and Aczel, in order to handle *schematic* and *multiple* consequences.

We focus on a particular Logical Framework, namely the *Calculus of Inductive Constructions*, originated by Coquand and Huet, and its implementation, Coq. Our investigation shows that this framework is particularly flexible and suited for reasoning on properties of programs and systems.

Our work could serve as a guide and reference to future users of Logical Frameworks.

Keywords: Computer-aided formal reasoning; Logical Frameworks, CIC, Coq; Formal methods, Program verification; Axiomatic, operational and natural semantics; Dynamic Logics, Modal logics, μ-calculus; Consequence relations.

March 1997

Address: Corso Italia 40, 56125 PISA, Italy. Tel:+39-50-887000 – Fax: 887226
mailto:miculan@dimi.uniud.it http://www.dimi.uniud.it/~miculan

Actual author's address:

Marino Miculan
Dipartimento di Matematica e Informatica
Università degli Studi di Udine
Via delle Scienze, 206 (loc. Rizzi)
33100 Udine (Italy).

Voice: +39 432 558456; Fax: +39 432 558499.
E-mail: `miculan@dimi.uniud.it`
WWW: `http://www.dimi.uniud.it/~miculan`

Copies of this thesis may be obtained by writing to:

Biblioteca
Dipartimento di Informatica
Università degli Studi di Pisa
Corso Italia, 40
56100 Pisa (Italy).

Fax: +39 50 887226.
E-mail: `tecrep@di.unipi.it`

To the thoughts
Of your heart
The only soughs
They'll never garth.

Contents

List of Figures

Acknowledgements

First of all, I wish to thank my supervisor, Furio Honsell, for his helpful suggestions and discussions along (and before) the years of my PhD courses. This thesis would never have seen the light of day, without his invaluable effort in keeping my (often diverting) attention on the subject of Logical Frameworks, resisting to the bewitching siren song of fleeting research issues.

I took full advantage of the many useful comments of my referee committee. I thank Eugenio Moggi and Carlo Montangero (and the aforementioned Furio) for their forbearing spurs and suggestions, which allowed me to find the narrow (and feasible) path in the secret dark wood of the far too ambitious starting proposal. I wish to thank Joëlle Despeyroux and Robert Harper, whose careful readings of the preliminary version of the thesis allowed me to greatly improve the quality of the presentation.

In the fertile environment of the Department of Computer Science of the University of Pisa, I had deep insights in many important research areas, many of which I not even skimmed before. Among the people I met there (too many to mention here), I like to thank the friends Daniela Archieri, Fabio Gadducci, Stefano Guerrini, Francesca Levi, Simone Martini, Andrea Masini, Angelo Monti, Paola Quaglia, Vladimiro Sassone.

A special regards is devoted to the giant figure of Ennio De Giorgi; the short time I had to know him then, has been enough to mourn his lack now.

I have completed my PhD course at the Department of Mathematics and Computer Science of the University of Udine, where I am grateful to Fabio Alessi, Pietro Di Gianantonio, Agostino Dovier, and Stefano Mizzaro for their help and support. In particular, Fabio's notes removed many categorical doubts (and tuned us in perfect fifths). Of course, I cannot forget to thank the local and anonymous TEX guru—with whom I have reached unfathomable abysses of schizophrenic insanity.

I benefited by a lot of suggestions by colleguas from all around the world. Among the many others I can't recall now, I mention Arnon Avron, Stefano Berardi, Peter Dybjer, Eduardo Gimenez, Frank Pfenning, and the whole *TYPES* community. I would also thank the INRIA CROAP group, for giving me the possibility of the exciting but unfortunately short stay in Sophia-Antipolis, during November 1994.

No pòs dismenteâ la mê famèe, ch'à mi à sostignût, e no dome economicamenti, durant i ains da universitât e dal dotorât.

Finally, my mind gets ashore to Lise, who learned how to steer my stem through the stormy tides of my moody sides. From snow-white ski tracks, to brick-red tennis courts, from sandy sunny beaches to grape-scented hills, she taught me that life has to be lived twice. At least.

Chapter 1

Introduction and Motivations

1.1 Introduction

Ever since the dawn of the computer era, the activity of programming and system designing has been recognized as particularly difficult. An utmost important task of this activity is providing *certifications*, i.e. proving whether the developed program/system meets its specifications. Ordinary programmers become convinced of the correctness of their programs through informal arguments, or by testing their behaviour on some sample input data. This approach is not complete: it cannot ensure the absence of errors under all circumstances. Often, this is not sufficient, especially in life-critical situations (such as on-board software, or high environmental risk plants): in these cases, programs and systems have to be formally proved to meet their specifications, i.e. *certified*. This can be achieved—at least theoretically—by using *rigorous* verification techniques: mathematical methods are used to *prove* that a program satisfies some properties, i.e. the *specifications*. Practically, however, even rigorous verification can only increase our possibilities that the program is correct. Not only our metatheory can have a flaw (quite unlikely) but (more likely) the justification that a set of specifications captures correctly a given "real environment" cannot be formally proved and ultimately rests on an act of faith.

The formal methods of mathematics have been developed on abstract mathematical entities. Analysts have their Banach spaces and topologies, algebraists have their groups and rings (*very abstract* algebraists have their categories and topoi), differential geometricians have their homological groups, quantum mechanic physicists have their Hilbert spaces, and so on. These structures are the objects of mathematical reasoning (the *models*): mathematicians reason on these models by using a rigorous, although *informal*, language; that is to say, the proofs that they carry out on these models are in that highly-specialized part of natural language which is spoken only in mathematical discussions.

Semantics of programming languages. The crucial point of the formal study of programs and systems is that *they can be given a (mathematical) semantics*, that is, one can define a mathematical model on which the formal methods of mathematics can be used to make a rigorous correctness argument. The purpose of such a mathematical model is to serve as a basis for understanding and reasoning rigorously about the behaviour of programs, programming languages, and systems. Almost every programming language in use has been subject to this kind of analysis. Semantics definition methods are useful both

to implementors and to programmers, for they provide:

1. a precise standard for implementations. The standard can be used as a reference for implementing the language in the same way on all machines;
2. useful user documentation. Every programmer *must* figure out the meanings of program constructors he is going to use, at some level of abstraction and detail. A formal semantic definition can help a trained programmer to develop his "language model"; he can read the definition and use it as a reference;
3. a tool for design and analysis. Typically, for studying the pragmatics of a language, we have to implement it. This is because there are few tools for testing and analyzing a language. Semantics definitions can suggest efficient and elegant implementations, just after having defined the syntax of the language. Actually, the designer of a language should take seriously into account how the meanings of programs will be best described;
4. input to a compiler generator. A compiler generator maps a semantics definition to a guaranteed correct implementation for the language. This frees the implementors from the most risky aspects of implementation.

The development of methods for best describing the semantics of a language is one of the primary challenges of Computer Science. Since the first '60s, many kinds of semantics have been introduced, but none has been accounted for the ultimate or best one. Traditionally, the semantics of a programming language can be given in three fashions:

- the *operational semantics* method describes the meaning of a programming language by specifying an *abstract machine* for the language. The operational semantics of a program is the evaluation history (i.e. the sequence of configurations) that the abstract machine produces in evaluating it. Operational semantics descriptions specify either the single transition step of this abstract machine (*small-step semantics*; see, e.g., [Plo81]), or the whole "evaluation process" in a single assertion (*big-step semantics*; see, e.g., [Kah87, Des86, BH90, Han93]);
- the *denotational semantics* method maps directly a program to its meaning, its *denotation*. Denotations are usually mathematical objects belonging to a suitable domain of meanings. Hence, the denotational semantics is a function from programs to denotations (see e.g. [Gun92, Sch86]);
- the *axiomatic semantics* method does not define explicitly meanings of programs. Instead, a method for deriving properties about programs is given. Usually, this method consists of a logical system, or calculus, for deriving formulæ which represent the properties we are concerned with. The meaning of a program could be defined, therefore, as the set of formulæ we can *formally* prove in the logical system. The character of the logical system is determined by the kind of properties that can be proved (see e.g. [Apt81, Sti92]).

Each of these three styles has its purposes. A clear operational semantics is very useful in implementation (we just need to realize the abstract machine for obtaining a correct interpreter), but its low abstraction level does not allow for powerful proof techniques; we can reason only by induction on the length of computations. A denotational semantics is

more abstract than an operational definition, for it does not specify computational steps. Its high-level, modular structure makes it especially useful to designers and programmers. However, the underlying mathematical theory may be very complex, and its implementation is not trivial. Axiomatic definitions are even more abstract than denotational ones, and can give rise to very elegant proof systems, useful in developing as well as verifying programs. However, they do not provide insights into the implementation, and even their expressive power may be not sufficient for characterizing completely the meaning of a program: there can be two programs, which behave differently, but the properties formally proved about them are the same.

Formal versus informal reasoning. It is worthwhile stressing the difference between a *rigorous* but *informal* proof and a *formal* proof. A *rigorous, informal* proof is a detailed and (hopely) clear step-by-step argument written in natural language, although highly specialized and tailored to a restricted audience of specialists. A *formal* proof instead is some syntactic expression, formulated in the syntax of a particular proof system (*formal system*) according to its axioms and rules.

Although it is believed that all informal proofs can be converted, more or less easily, into formal proofs of some logic, such as a higher-order set theory, mathematicians still prefer the formers in reasoning about their abstract models. The point is that in Mathematics the *models* are the primary objects of attention, while formal systems are just tools, not even essential. Therefore, *semantic* arguments, based on the abstract properties of models, are the norm in mathematics, while formal argument are absolutely rare.

On the other hand, since the late '60s formal methods for proving properties of programs and programming languages have been given an utmost importance. There are many reasons for this.

First, the objects of attention of computer science are programs and programming languages, and programs are themselves formal objects. A formal system can hence reason directly about programs by embedding the programs themselves in the formulæ. An example of this approach are the well-known *Hoare logic* and *proof outlines*, where programs can be "annotated" by assertions which can be used in a step-by-step verification of the program itself [Apt81, AO91]. Moreover, modern operational semantics use formal systems to specify either the transition step of an abstract machine (such as in the *Structured Operational Semantics* [Plo81]) or more abstractly the "evaluation notions" associated to the language (e.g. *Natural Semantics, Natural Operational Semantics, Extended Natural Semantics* [Kah87, BH90, Mic94, Han93]).

Secondly, many times semantic arguments are not satisfactory because there is no direct connection between the abstract notions of the model, and the properties we would like to prove on the program. Often, there is even a poor understanding of *what* a semantics *is*. The point here is that in Computer Science, differently from Mathematics, semantical models are no more the central objects of investigation, but just *tools* for reasoning about programs. The definition of a "good" model is therefore committed to the kind of properties we want to prove. Often, although we know perfectly the kind of properties we have to deal with, the definition of an adequately expressive model is complex and very subtle. This is the case, for instance, of many concurrent languages: a plethora of semantic structures for concurrency have been proposed so far, but no one of them is considered to be the best and ultimate one. Moreover, the richer is the language, the

more complex is the mathematical theory on which its semantics relies; programmers and language designers are seldom familiar with complex mathematical theories. In such cases, a common way for avoiding the difficulties and subtleties of an obscure or unsatisfactory model is to drop semantic arguments and to resort to some formal system (proved sound for the semantics, once and for all) which proves directly the properties we need.

A third motivation is that computer programs get easily large and complicated. Informal but rigorous proofs of correctness become therefore long and blurred by tedious lists of cases. The human prover is hence prone to oversights and omissions, and large informal proof may be difficult and obscure to the reader. Formal systems may provide a means for managing large proofs: the human prover is urged for more detailed and clear proofs, and the reader is less likely inclined to misunderstandings.

Computer Aided Formal Reasoning. It should be clear, however, that formal proof systems are not the *panacea* in proving properties of programs and programming languages. The definition of a complete formal system is not always easy; it may be as difficult as defining a satisfactory semantical model. Usually, more than just one formal system arise for a given particular programming languages, according to which semantical properties we focus on (e.g. partial/total correctness, equivalences of programs, etc.). Even, the same properties can be often described in many different ways, giving raise to many formal proof systems. Moreover, a formal, syntactic proof is often crammed with routine details which are often neglected, if not "swept under the rug," in informal proofs. Hence, the verification and analysis of properties becomes error-prone and not easily applicable "by hand," so that complete rigorous arguments of correctness seldom accompany large programs.

Unfortunately, due to intrinsic complexity and undecidability limitations, the automatic verification of a condition is seldom feasible. Most of the times, the properties we face in program verification are not decidable (e.g. the termination problem), or the complexity of decision procedures is infeasible. Therefore, a completely automatical environment, such as a *theorem prover*, for proving properties on programs in general is not feasible. Nevertheless, in some restricted cases it is still possible automatize (part of) derivations of properties, freeing the human from a difficult and unpleasant chore.

Otherwise, the problem can be (partially) overcome by supplying the human verifier with a valid aid for reasoning with these formal systems. The verification process carried out by human certifier can be assisted by some semi-automated aids, which support an interactive development of error-free proofs. What is needed is a *computer-aided proof environment*, that is, a system in which we can represent (*encode*) the formal system, more or less abstractly: its syntax, axioms, rules and inference mechanisms. After having supplied the proof environment with a representation of the formal system, the human user should be able to correctly manipulate (the representations of) the proofs: the environment should provide tools for

- checking previously hand-made proofs;
- developing interactively, step-by-step, error-free proofs from scratch;
- reusing previously proved properties;
- even, deriving properties automatically, when feasible, freeing the human from most unpleasant and error-prone steps.

As we have pointed out before, usually many formal proof systems arise for a particular logic. However, the implementation of a proof environment for a specific formal system is a complex, time-consuming, and daunting task. At the level of abstract syntax, support has to be provided for the management of variables, binding operators, substitution, and schemata of formulæ, terms, and rules. At the level of proofs, a representation of formal proofs must be defined, and mechanisms associated with proof checking and proof construction must be provided. This includes the means for instantiating rule schemata, and constructing proofs from rules, checking the associated context-sensitive applicability conditions. Further effort is needed in order to support automated search: tactic and tacticals, unification, matching, search startegies have to be implemented. Therefore, a uniform and reliable alternative is highly desiderable.

A possible approach is to observe that any (program) logic can be translated into a logic with enough expressive power, such as First Order Logic (FOL), Higher Order Logic (HOL), or Set Theory (ZF). In these logics, in fact, one can formalize the *semantics* of the object logic by axiomatizing the abstract notions about the model. Informal arguments on the semantics of the object logic can be then carried out as formal proofs of FOL (or ZF). For instance, we can straightforwardly translate Temporal Logic into FOL just by introducing in every formula a "temporal variable" which represents the instant in which the formula holds. Every formula φ of Temporal Logic is translated into a corresponding first-order formula $\varphi'(t)$; in particular, the translation of $\Box\varphi$ is $(\Box\varphi)'(t) = \forall t'.(t' \geq t \supset \varphi'(t'))$. Of course, we need to axiomatize the flow of time, by adding suitable assumptions on the temporal variables.

Hence, one may argue that implementing a proof editor for a single very general logic, such as FOL, HOL, or ZF, would be enough for encoding any other program logic. This is true, and indeed this approach is pursued by several projects (see e.g. the *Isabelle* and the *HOL* systems [NP92, Gor88, Gor89]), but we achieve this simplification at the price of clarity and simplicity. In fact, in these "artificial" encodings one formalizes and focuses on the semantical notions of the logics, instead of dealing with its formal proof systems. Hence, the universe of discourse has to be circumscribed with the whole axiomatisation of the domain, introducing cumbersome hypotheses which have to be dealt with. Although the analogy is not perfect, FOL or ZF may be compared to an assembler language: we can express everything within them, but in general in a complicated way. This is the reason for preferring more specific and "high-level" logics (similarly to "high-level languages"). These logics are not as general as FOL or ZF, because they are tailored for the specific case we are facing, but they are much easier to use and understand. Indeed, this is the main reason for which there are so many different logics: in each of them, we sacrifice generality for achieving conciseness and simplicity, by focusing on some important notions. In Temporal Logics, for instance, we make the "temporal variable" more important than the other individual variables by adopting special formulæ constructors which make the dependency on the time implicit.

Logical Frameworks. A more promising solution is to develop a general theory of logical systems, that is, a *Logical Framework*. A Logical Framework is a metalogical formalism for the specification of formal systems of logics. This approach differs substantially from the previous one: while general-purpose logics can be used in specifying semantical notions, Logical Frameworks are formalisms for specifying the *syntactic, deductive* notions

of logics. Indeed, a key feature of Logical Frameworks is that they always provide suitable means for representing and deal with, in the metalogical formalism, the *proofs* and *derivations* of the object formal system.

As for any other specification language, a Logical Framework should be as simple and uniform as possible, yet it should provide means for expressing concisely and faithfully the uniformities of a wide class of formal systems, so that much of the implementation effort can be expended once and for all. Indeed, the implementation of a Logical Framework yields a *logic-independent proof development environment*. Such an environment must be able to check validity of deductions in a formal system, after it has been provided by the specification of the system in the formalism of the Logical Framework.

Although we are still at the dawning of the research about Logical Frameworks, several different frameworks have been proposed, implemented and applied to many formal systems. In recent years, *type theories* have emerged as leading candidates for Logical Frameworks. Simple typed λ-calculus and minimal intuitionistic propositional logic are connected by the well-known *proposition-as-types* paradigm [Chu40, dB80]: with this interpretation, a proposition is viewed as a type whose inhabitants correspond to proof of this proposition. Stronger type theories, such as the *Edinburgh Logical Framework* (ELF) [HHP93, AHMP92], the *Calculus of Inductive Constructions* (CIC) [CH88, Wer94] and *Martin-Löf's type theory* (MLTT) [NPS92], were especially designed, or can be fruitfully used, as a logical framework. In these frameworks, we can represent faithfully and uniformly all the relevant concepts of the inference process in a logical system: syntactic categories, terms, assertions, axiom schemata, rule schemata, tactics, etc. via the *judgements-as-types, proofs-as-λ-terms* paradigm [HHP93]. The key concept is that of *hypothetico-general* judgement [Mar85], which is rendered as a type of the dependent typed λ-calculus of the Logical Framework.

It is worthwhile noticing that Logical Frameworks based on type theory directly give rise to proof systems in *Natural Deduction style* [Gen69, Pra65]. This follows directly from the fact that the typing systems of the underlying λ-calculi are in Natural Deduction style, and rules and proofs are faithfully represented by λ-terms, following the judgements-as-types paradigm [HHP93]. As it is well-known, Natural Deduction style systems are more suited to the practical usage, since they allow for developing proofs the way mathematicians normally reason.[1]

These type theories have been implemented in logic-independent systems such as Coq [CCF+95], LEGO [LPT89], and ALF [MN94] (we recall also the language *Elf* [Pfe89], which adds to the type-theoretic approach of the Edinburgh Logical Framework the proof search machinery of logic programming). These systems can be readily turned into interactive proof development environments for a specific logic: we need only to provide the specification of the formal system (the *signature*), i.e. a declaration of typed constants corresponding to the syntactic categories, term constructors, judgements, and rule schemata of the logic. It is possible to prove, informally but rigorously, that a formal system is correctly, *adequately* represented by its specification in the Logical Framework.

[1] Indeed, the Natural Deduction style has been introduced by Gentzen because it "reflects as accurately as possible the actual logical reasoning involved in mathematical proofs" [Gen69].

1.2 Aims of this thesis

As soon as a human program verifier, overwhelmed by lots of tedious and error-prone details in using formal systems "by hand", discovers the flexibility and potentialities of Logical Frameworks, he wonders whether these tools can simplify his task.

A deep investigation of the applicability of Logical Frameworks, both theoretical and practical, for encoding formal systems of programs is in order. Bearing in mind the problems we encounter in reasoning about programs, many questions naturally arise, including (but not limited to) the following:

- can Logical Frameworks be fruitfully applied in reasoning about properties of programs (such as correctness, termination, equivalences, etc.)?
- how to deal with the problems one will face in encoding a program logic in a Logical Framework (such as representation of program variables, implementation of non-standard substitutions, etc.)?
- what can be delegated to the metalanguage, taking advantage of the metalogical features of Logical Frameworks (e.g., schematicity of formulæ and rules, substitution via *higher-order abstract syntax*)?
- among many encoding solutions, what is the best suited for a particular logic? and on the other hand, what is the "best formulation" (Hilbert-, Natural Deduction-, sequent-style) for a program logic, for being represented in a Logical Framework?

This thesis aims to address these questions. The main aim is to streamline the process of encoding logical theories of languages and programs, in type theories (namely, in the Calculus of Inductive Constructions), understanding how and at what extent we can take advantage of the features of the metalanguage. Many problems that generally arise in formalizing program logics in Logical Frameworks will be described, discussing and comparing possible solutions.

As for any logic and proof system, in formalizing a formal system for a programming language, we can isolate two parts. The first is the *theory of expressions*, that is the (abstract) language of the formal objects which we will deal with: programs, terms, formulæ, etc. The second part is the *theory of proofs*, that is the formal systems representing the semantics of the language.

In the area of formal systems for programs and programming languages, both these parts often present many idiosyncrasies which clash with the structural properties of the metalanguage of Logical Frameworks; these include (but are not limited to) negative formulæ constructors, complex notions of instantiation and substitution, context-sensitive grammars, typing systems,[2] infinitary formulæ, polyadic binding constructors, subsorting, equivalence theories of expressions, infinitary rules, rules with weird side conditions, and so on. Most of the times, these anomalies escape the "standard" representation paradigm of Logical Frameworks.

The solution proposed in this thesis is that in order to get the best result, both aspects of Logical Frameworks and of formal systems need to be faced. On the hand of Logical

[2] These conditions are sometime called *static semantics*.

Frameworks, we need to develop and investigate new efficient representation techniques. On the other hand, the metalogical features of Logical Frameworks have a retrospective effect on the design and development of formal systems themselves. Logical systems given *a priori*, without bearing in mind their usage in a proof assistant, may be of difficult encoding and usage, pratically. Often, a complete reformulation is needed, in order to simplify the encoding process and the subsequent usage. This reformulation process will interest both the syntactic and the deduction part of the formal system.

Since generated editors allow the user to reason "under assumptions", the designer of a proof editor for a given logic is urged to look for a formulation of the logic which can take best advantage of the possibility of manipulating assumptions. This study takes us into truly metatheoretic and encoding-independent issues, such as the investigation of the crucial concept involved in discussing the notion of assumption for a given logic, i.e. the notion of *consequence relation*. Consequence relations are formal representations of logical dependencies between assumptions and conclusions. They play a crucial rôle in stating and proving the adequacy of encodings in Logical Frameworks. Usually, a logic gives rise to more than one consequence relation. Before building an editor for a given logic, the designer/implementor has to clarify two equally important, apparently orthogonal, issues.

- Which consequence relation is the one to focus on?
- Which style of presentation is best for actually "using" the logic, e.g. Hilbert, Natural Deduction or Gentzen (sequent) style?

In the methodology of Logical Frameworks, answering the first question amounts to decide which *judgements* to encode. Experience shows that Natural Deduction style systems are best suited for exploiting the reasoning power of assumptions provided by Logical Frameworks. Therefore, we will emphasize the importance of assumptions by introducing new proof systems, in Natural Deduction style, for the logics presented in this thesis. These Natural Deduction-style systems allows us also to introduce another neglected metatheoretic issue, that of *proof-theory* of program logics.

1.3 Motivations

The aims of this thesis are both theoretical and practical.

As we have already pointed out before, an encoding can be immediately embedded in a proof environment, providing a human verifier/developer of programs of a computerized interactive proof editor. Even, in the case that decidability and complexity of proof search is not prohibitive, it is possible to utilize the encoding as a logic program in a type-theory based theorem prover, such as Elf [Pfe89]. For instance, a specification in the Edinburgh Logical Framework of an operational semantics yields directly a *prototype interpreter* for the specified language [MP91, Mic94].

An objection one may raise to this approach is that proof editors generated by means of Logical Frameworks are not so efficient as some *special-purpose editors*, which are those especially designed and tailored to a specific logic. Nevertheless, Logical Frameworks can be very useful for many theoretical and practical reasons.

The first, immediate use of Logical Frameworks is as "logic specification languages", independently from any practical application. This has several consequences. The encoding

in a Logical Framework often provide the "normative" formalization of the formal system under consideration. The specification methodology of Logical Frameworks, in fact, forces the user to make precise all tacit, or informal, conventions, which always accompany any informal presentation of a logic.

Secondly, Logical Frameworks provide a common medium for integrating different systems into the same formalism. From a theoretical point of view, this allows for investigating the connections and similarities between systems which are usually kept disjoint. From a practical point of view, generated editors rival special-purpose editors when efficiency can be increased by integrating independent logical systems. This is not possible in special-purpose editors, since they are not easily extendible by adding other systems beside the one they implement.

Thirdly, the interactive environments obtained by specifications are *natural*, in the sense that they implement the formal system, and not some other presentation, of the original logic. The same derivations and proofs one can carry out with pencil and paper, can be carried out in the proof environment, interactively and with no possibility of errors. A user of the original proof system can transfer immediately to these implementations his practical experience and "trade tricks." Therefore, he is not forced upon by the overhead of unfamiliar or indirect encodings, as would editors derived from, say, FOL editors via an encoding. As we have already pointed out, these latter editors force the user to deal with semantics aspects of the logics, and which do not belong to the original formal system.

Moreover, the wide conceptual universe provided by Logical Frameworks allows, on various occasions, to device genuinely new presentations of formal systems. As we have pointed out before, one of the main improvement that Logical Frameworks based on Type Theory directly give rise is the introduction of proof systems in Natural Deduction style. This yields both theoretical and practical advantages. From the theoretical point of view, these presentations are interesting independently from their encoding/implementation, since they give us the possibility of exploring metatheoretic issues of program logics, which usually are neglected (proof theory, infinitary completeness, multiple consequence relations, etc.). From the practical point of view, generated editors allow the user to reason "under assumptions" and go about in developing a proof the way mathematicians normally reason: using hypotheses, formulating conjectures, storing and retrieving lemmata, often in top-down, goal-directed fashion.

Beside these practical and theoretical motivations, this work can give insights in the expressive power of Logical Frameworks. The field of Computer Aided Formal Reasoning is a very active area, and Logical Frameworks and their implementations are still under development. This work can elighten strong and weak points of the use of Logical Frameworks and their implementations (more specifically, of CIC and Coq) in connection with program logics, giving suggestions for their improvement. A great effort is also put into the implementation and the development of user friendly interfaces proof editor based on Logical Frameworks. This thesis can give insights in the pragmatic problems connected with the use of these environments.

1.4 Synopsis

The thesis consists of four parts.

1.4.1 Part I: General Metatheoretic Issues

The first part is devoted to the analysis of the basic concept which are involved in defining the two main components of a logic, i.e. the syntax of the language and the consequence relation. This analysis will be useful in investigating the properties of the logical system in view of their implementation/specification (Parts II, III).

In Chapter 2 we recall briefly a general theory of expressions, inspired by Martin-Löf theory of arities. In particular, we recall the principle of *higher-order abstract syntax*, which is a fundamental approach to the treatment of binding operators and bound variables.

In Chapter 3 we provide an introduction to *consequence relations* as the main metatheoretic tool in investigating the entailment relation established by semantics and proof systems. They are also of great releavance in stating and proving the adequacy of encodings of proof systems in Logical Frameworks. Classic approaches to consequence relations (e.g. Avron' *simple consequence relations* and Aczel' *schematic consequences* [Avr91, Acz94]) do not fit very well to the proof systems of Part II. The logics we deal with, indeed, present some uncommon features, such as nonstandard notions of substitutions or proof systems which allow for infinitary derivations and deal with many entailment notion between formulæ at the same time. In this chapter we introduce the semantical counterpart of these systems, namely the *multiple, infinitary consequence relations*. Finally, we will extend the standard notions of substitution by introducing the *nondeterministic substitution schemata*.

1.4.2 Part II: The Object Logics

In this second part, we study the style of Natural Deduction and the possibility of presenting in ND-style the logical systems of a set of paradigmatical logics of programs:

- Structural and Natural Operational Semantics (Chapter 4);
- Modal Logics (Chapter 5);
- Propositional Dynamic Logic (Chapter 6);
- First Order Dynamic Logic (Chapter 7);
- Hoare Logic (Chapter 8);
- μ-calculus (Chapter 9).

The study of the above systems leads us to address problematic issues such as

- the presence of "modal" formula constructors and proof-rules;
- the presence of infinitary proof rules;
- complex notions of instantiations, due to the duality of identifiers used both as program and logical variables;
- the presence of negative formulæ constructors;
- the presence of context-sensitive conditions on the applicability of syntactic constructors.

We will address and discuss solutions to these problems, which arise when facing most program logics. The solutions we will propose for these problematics are orthogonal, in the sense that they can be applicable independently from each other.

Many of these issues (such as the definition of Natural Deduction style proof systems, discussions on various kinds of modalities, on infinitary systems, comparison of several notions of completeness) are essentially theoretical, and indepented from the fact that we are going to encode these systems. Therefore, we have preferred to treat them on their own, keeping apart from the issues related to the encoding paradigm.

In Chapter 4 we will study, from a logical point of view, how to take advantage of the Natural Deduction style for representing concisely operational semantics. As we will see, this has both theoretical and practical advantages. This study yields the introduction of the *Natural Operational Semantics* style of presentation, and the *bookkeeping technique*. We discussion in detail the applicability and expressive power of this technique; we present a complete case study by examining a functional language extended with truly imperative features such as assigments and procedures.

Apart from Chapter 4, each of the later chapters of this part is structurated in the same manner, as follows:

- Definition of syntax and semantics of the logic under consideration;
- Definition of the semantic counterpart of proof systems, that is, the consequence relations for the logic and the related completeness notions;
- Definition of Natural Deduction style proof systems for representing the introduced consequence relations, with proofs of their adequacy;
- Related work, proof theory, and applications.

1.4.3 Part III: The Theory of Formal Representation

In the third part, we will discuss in detail the encoding of the systems presented in the previous part.

We begin the part with an introduction to Logical Frameworks based on type-theory (Chapter 10). In this chapter, we recall the class of *Pure Type Systems*, introducing also a Gentzen-style version of PTS's (Section 10.1). A paradigmatical Logical Framework is the *Edinburgh Logical Framework*, which we present in Section 10.2. In Section 10.3 we briefly recall the features of the logical framework we will use subsequentally, that is, the *Calculus of Inductive Constructions*.

In the following chapters, we will face–in an incremental manner–the problematic issues of encoding program logics. Each chapter will focus on specific problematic issues, addressing some possible solutions; the techniques and solutions presented at some point can be used in the following, whenever needed.

We will start from a detailed account of the "judgements-as-types" paradigm (Chapter 11). In particular, we examine the feasibility of *higher-order abstract syntax* and *inductive definitions*. We will see that, although they are usually adopted in paradigmatical encodings, their application in the case of program logics is very limited. We will discuss some possible solutions. Finally, we briefly discuss other problems which arise in connection with context-sensitive conditions on languages, and in presence of subsorts.

In Chapter 12 we will deal with the encoding of operational semantics, in particular the Natural Operational Semantics paradigm.

In the following chapters we focus on the representation of complex properties of proof systems in the Calculus of Constructions.

- in encoding Modal Logics we discuss how to represent modalities and proof rules (Chapter 13);
- in encoding Propositional Dynamic Logics we focus on the representation of infinitary proof systems (Section 14.1);
- in encoding (First-Order) Dynamic Logics we focus on the problems arised by the dual nature of identifiers, and the clash between logical and program variables (Section 14.2);
- in encoding the μ-calculus, we discuss the problems arised by context-sensitive conditions on formulæ, and negative formula constructors in proof systems (Chapter 15).

In each of these chapters we will present the encoding signature and the related theorems which prove its adequacy with respect to the represented logic.

A complete, exhaustive discussion of *every* problem one may encounter in view of the specification of *any* program logic is out of the scopes of this thesis; for instance, we will not treat polyadic bindings, or discuss the logical representation of denotational semantics. We will briefly recall these and some others possible future works in Chapter 18.

1.4.4 Part IV: Pragmatics

The fourth and final part is devoted to the discussion of some pragmatical issues concerning the experiments carried out in the proof environment Coq. We will describe the implementation of finitary and infinitary proof systems for Dynamic Logic (Chapter 16) and the implementation of μ-calculus (Chapter 17). Some examples of derivations carried out formally in the machine are described. These examples include the derivation of Hoare Logic in Dynamic Logic, and the proof of equivalence between some simple programs.

Prerequisites.

No particular knowledge is presupposed for Part I.

Some knowledge of the operational and axiomatic semantics of programming languages, and of the Natural Deduction style, is advisable for reading Part II.

Some knowledge of the theory of typed λ-calculi, of Logical Frameworks and specifically of the Calculus of Inductive Constructions would be useful in reading Part III.

Finally, Part IV is addressed to whom is concerned by the pragmatic issues of the specific Coq proof environment. A good knowledge of the Coq system is advisable.

Part I

General Metatheoretic Issues

Chapter 2

A theory of arities

In this chapter, we give a general description of the language of an object logic. We follow Martin-Löf's theory of arities [NPS90]. It is worthwhile noticing that, from our point of view, syntactic objects are labelled trees (parsing trees), and not strings of symbols.

Definition 2.1 (Syntactic Sorts and Variables) *A set of symbols $\mathcal{C}$ is a set of* (syntactic) sort constructors *if each symbol $c \in \mathcal{C}$ is associated to a natural number n; we denote this fact by $c^n \in \mathcal{C}$. The number n is said the* arity *of c.*

The set Sorts of (syntact) sorts *on a set $\mathcal{C}$ of sort constructors is defined inductively as follows:*

- *0-ary sorts constructors are syntactic sorts;*
- *if $c^n \in \mathcal{C}$ and $S_1, \ldots, S_n$ are sorts, then $c(S_1, \ldots, S_n)$ is a sort.*

A sort S contains variables *if it is associated to a set Var^S of countably infinite symbols. Each element $x \in Var^S$ is called a* variable (of sort S)*, and it is said to* range over *the sort S.*

Notice that a sort may contain no variables; this is the case, for instance, of the sort of formulæ in first order logic.

Definition 2.2 (Arities) *The* set of arities *Arit on Sorts and their* levels *are defined inductively as follows:*

- *each sort $S \in Sorts$ is an arity; its level is 0;*
- *for $\alpha_1, \ldots, \alpha_n \in Arit$ and $S \in Sorts$, then $(\alpha_1, \ldots, \alpha_n) \to S$ is an arity; its level is $1 + \max\{\alpha_1, \ldots, \alpha_n\}$.*

Definition 2.3 (Expressions) *A set of symbols $\mathcal{E}$ is a set of* expression constructors *for the sorts Sorts if each symbol $e \in \mathcal{E}$ is associated to an arity $\alpha \in Arit$ on Sorts; we denote this fact by e^α.*

The set of expressions *Expr defined by $\mathcal{E}$ for the sorts Sorts, and their arities, is defined inductively as follows:*

- *if x is a variable of sort S then x is an expression of arity S.*

- *for $S \in Sorts$ and $\alpha_1, \ldots, \alpha_n \in Arit$ if $f \in \mathcal{E}$ has arity $(\alpha_1, \ldots, \alpha_n) \to S$, and $e_1, \ldots, e_n$ are expressions or arity $\alpha_1, \ldots, \alpha_n$, respectively, then $f(e_1, \ldots, e_n)$ is an expression with arity S.*
- *if e is an expression of arity S of level 0, and $x_1, \ldots, x_n$ are variables of arities $S_1, \ldots, S_n$ (of level 0) respectively, then $(x_1, \ldots, x_n)e$ is an expression of arity $(S_1, \ldots, S_n) \to S$.*

In the last clause, any free occurrence of $x_1, \ldots, x_n$ in e is bound. *Two expressions which are equal up to renaming of bound variables are the same expression (α-equivalence).*

Let $X = \{x_1^{S_1}, \ldots, x_n^{S_n}\}$ a finite sequence of variables. We denote by Expr_X the set of expressions with free variables in X; for $S \in Sorts$, we denote by S_X the set of expressions of arity S with free variables in X.

It is worthwhile noticing that every binding constructor of any object language is defined formally as an expression constructor of arity greater than 0. This allows us to take care of many binding constructors uniformly, by delegating every object-level α-conversions to the only α-equivalence of higher-order arities.

Example 2.1 Consider the following standard first-order language:

$$\begin{array}{rrcl} Term: & t & ::= & 0 \mid x \mid s(t) \mid t_1 + t_2 \\ \Phi: & \varphi & ::= & \mathit{ff} \mid t_1 = t_2 \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \forall x\varphi \end{array}$$

where x ranges over a set of first-order variables, *Var*.

This language is composed by two sorts: the sort of terms, which contains variables, and the sort of formulæ, which does not. Therefore, this language can be fully described by specifying:

- the sort constructors: $\mathcal{S} = \{Term^0, \Phi^0\}$. In this case, $\mathcal{S} = Sorts$ because there are no higher-order sort constructors;
- the sorts with variables: $\mathcal{V} = \{Term\}$. The associated variable set is *Var*;
- the expression constructors, each with its arity.

Constr.		**Arity**	**Constr.**		**Arity**
0	:	$Term$	ff	:	Φ
s	:	$Term \to Term$	$=$	:	$(Term, Term) \to \Phi$
$+$	:	$(Term, Term) \to Term$	$\neg$	:	$\Phi \to \Phi$
			$\vee$	:	$(\Phi, \Phi) \to \Phi$
			$\forall$	:	$(Term \to \Phi) \to \Phi$

Hence, a formula of the form $\forall x.\varphi$ should be written more formally as "$\forall((x)\varphi)$", since $(x)\varphi$ is an expression of arity $Term \to \Phi$. Notice that the level of the arity of "$\forall$" is 1, while the other arities are of level 0. For this reason, "$\forall$" is called a *higher-order* constructor.

□

A complete description of the language of a logic is therefore obtained by specifying the sort constructors, the sorts with variables and the expressions constructors:

Definition 2.4 (Alphabet) *The* alphabet *of a logic is a triple* $\langle \mathcal{C}, \mathcal{V}, \mathcal{E} \rangle$ *where*

- $\mathcal{C}$ *is a set of sort constructors;*
- $\mathcal{V}$ *is a subset of the sorts on* $\mathcal{C}$*, denoting the sorts containing variables; for each* $S \in \mathcal{V}$*, let* Var^S *be the set of its variables;*
- $\mathcal{E}$ *is a set of (pairs of)* expression constructors *on the sorts on* $\mathcal{C}$*, together with their arities.*

The examples of object languages we shall deal with will be, in general, multi-sorted and will have higher-order expression constructors. The practice of representing binding constructors by means of higher-order expression constructors is called *higher-order abstract syntax* [PE88, HHP93, NPS90], and plays a fundamental rôle in the formal representation of languages (Chapter 10).

Chapter 3

Consequence Relations

An important notion in the presentation of a logic is that of *consequence relation*, also known as "logical consequence". A Consequence Relation is an abstract representation of the logical dependencies between assumptions and conclusions. This notion goes back to Tarski and was taken as the fundamental one by Gentzen [Gen69] and any subsequent treatment of Logics.

Consequence relations are important in studying proof systems which allow for reasoning under assumptions. In fact, these systems naturally provide a "consequence" notion, namely the one represented by derivability, that is "what follows from what". Hence, these systems cannot be completely understood without giving a semantic counterpart to the syntactic notion of "derivability".

The aim of this chapter is to give an insight into the main issues concerning consequence relations. In Section 3.1, we introduce the notion of Simple Consequence Relation; we discuss how a consequence relation can be defined in an abstract way (e.g. by means of a semantics, Section 3.1.1) and represented by a proof system (Section 3.1.2). In Section 3.1.3 we provide, as examples, a wide range of consequence relations which can be defined on Propositional Logic, First-Order Logic and Propositional Modal Logic.

In Section 3.2 we will discuss the notion of *schematicity*. This notion is particularly important for two reasons: firstly, logics are usually given by means of schemata (of rules and axioms), and secondly encodings of logics in type-based Logical Frameworks are built upon a particular kind of schematicity of rules (Section 11.1.)

In Section 3.3 we will generalize this definition to the notion of Multiple Consequence Relation. This generalization is needed in order to give an abstract interpretation of "multi-judgement" proof systems such as those arising in the encoding of many logics (such as modal logics, Dynamic Logics, operational semantics, etc.) in Logical Frameworks.

Finally, in Section 3.4 we briefly recall the Natural Deduction style of presentation of proof systems.

3.1 Simple Consequence Relations

We begin with a simple abstract notion of consequence relation, which will then generalize. Our definitions differ from, although they are inspired by, those of Avron and Aczel [Avr91, Acz94]. In the following, by "set of formulæ" we mean any r.e. (syntactic) set.

Definition 3.1 (Simple Consequence Relation) *Let Φ be a set of formulæ. A* (simple, single conclusioned) consequence relation *(SCR) $\rhd$ on Φ is a monotone, reflexive and transitive binary relation*[1] $\rhd \subseteq \mathcal{P}_{\mathrm{re}}(\Phi) \times \Phi$*:*

reflexivity: $\forall A \in \Phi : \{A\} \rhd A$*;*

transitivity: *for* $\Gamma_1, \Gamma_2 \in \mathcal{P}_{\mathrm{re}}(\Phi)$*,* $A, B \in \Phi$*, if* $\Gamma_1 \rhd A$ *and* $\Gamma_2 \cup \{A\} \rhd B$ *then* $\Gamma_1 \cup \Gamma_2 \rhd B$*;*

monotonicity: *for* $\Gamma_1, \Gamma_2 \in \mathcal{P}_{\mathrm{re}}(\Phi)$*,* $A \in \Phi$*, if* $\Gamma_1 \rhd A$ *then* $\Gamma_1 \cup \Gamma_2 \rhd A$*.*

In the following, we will drop the "set notation," and we will write A for $\{A\}$, Γ_1, Γ_2 for $\Gamma_1 \cup \Gamma_2 \ldots$

Some remarks are in order. Our definition 3.1 is different from both Avron's and Aczel's [Avr91, Acz94]. We consider *(possibly infinite) r.e. sets* of formulæ as antecedents and a *single* formula as consequence, instead of finite multisets on both sides [Avr91], or finite sets of assumptions [Acz94]; moreover, we require CR's to be monotone. We make these choices in view of the structural rules of the Logical Frameworks we focus on. In fact, as we will see, the structural features of Logical Frameworks based on type theories lead us to consider systems in Natural Deduction style [Pra65]. These systems are intrinsically monotone (they admit the "weakening" rule) and treat assumptions as sets. In these cases, therefore, the definition we have given has enough expressive power, without being overwhelmed by the treatment of multisets and other even more complex structures. Of course, there are many other logics (e.g. Girard's linear logic [Gir87a]) which cannot be fully understood without a finer treatment of assumptions, such as the one provided by multisets (or even more by *lists*, as Gentzen implicitly did [Gen69]). On the other hand, we allow for *infinite* sets of assumptions, since some of the systems we will introduce are infinitary. We require however these sets of assumptions to be recursively enumerable, and hence images of a primitive recursive function from $\mathcal{N}$ to Φ.

3.1.1 Free Consequence Relations from Semantics

There are two principal ways for defining consequence relations over a logic: *effectively* or *non-effectively*. In the first case, we provide a method for checking when a given formula is a consequence of a given set of assumptions. Typically, these methods are specified by means of *proof systems* (see the next subsection for more details.) Here, we focus on the latter case, that is when CR's are defined by means of non-effective characterization. Typical examples are definitions involving abstract mathematical structures, i.e. the *models* of the logic. These consequence relations are usually intended as *semantic* consequences. By convention, semantic consequence relations are denoted by "double turnstile" symbols, like "$\models$."

In the literature, these consequence relations have been defined semantically in many different ways. It turns out that most of them can be subsumed in the following general construction:

Definition 3.2 (Free Consequence Relation) *Let $\langle D, \sqsubseteq \rangle$ be a complete lattice, and $[\![\cdot]\!] : \Phi \to D$ a semantics for Φ in D; let $[\![\Gamma]\!] \stackrel{\mathrm{def}}{=} \sqcap_{A \in \Gamma} [\![A]\!]$.*

[1] At this point, we prefer the symbol $\rhd$, since $\models$ and $\vdash$ will denote particular kinds of consequence relations (respectively, those defined by means of semantic notions and proof systems).

The free consequence relation $\models$ on $[\![\cdot]\!]$ *is defined as* $\Gamma \models A \iff [\![\Gamma]\!] \sqsubseteq [\![A]\!]$.

It is easy to see that a free consequence relation is a simple consequence relation. Reflexivity and monotonicity are trivial. For transitivity, suppose $\Gamma \models A$ and $\Gamma, A \models B$. Then, by definition $[\![\Gamma]\!] \sqsubseteq [\![A]\!]$ so $[\![\Gamma]\!] = [\![\Gamma]\!] \sqcap [\![A]\!] = [\![\Gamma, A]\!] \sqsubseteq [\![B]\!]$ and hence $\Gamma \models B$.

We will see that most important consequence relations (such as, for instance, those of the systems of Part II) are in fact free CR's.

Remark 1. The adjective "free" is suggested by the categorical viewpoint. Indeed, the semantics $[\![\cdot]\!]$ induces naturally an axiomatic system $\langle \Phi, \{\langle \Gamma, A\rangle \mid [\![\Gamma]\!] \sqsubseteq [\![A]\!]\}\rangle$. The free consequence relation on $[\![\cdot]\!]$ is then the reflection of this axiomatic system, through the left adjoint of the inclusion functor of the category of Consequence Relations in the category of Axiomatic Systems (see [Acz94, Proposition 1], for the definition of these adjoints). Such reflections are usually called "free constructions" (e.g. free monoids from sets are reflections of sets in the category of monoids).

Remark 2. The need of complete lattices may be not so evident, at first. In the definition, instead of complete lattices we could consider meet semilattices just as well; however, since we need to give a semantics to infinite sets of formulæ, the image of $[\![\cdot]\!]$ has to be a complete meet semilattice. Now, a complete meet semilattice is always a complete lattice: just define $\bot \stackrel{\text{def}}{=} \sqcap [\![\Phi]\!]$ and $\sqcup A \stackrel{\text{def}}{=} \sqcap \{x \mid \forall y \in A. y \sqsubseteq x\}$. Therefore, the image of $[\![\cdot]\!]$ has to be a complete lattice even if D may be not. Now, points of D which are not in the image of the semantics are not significant; therefore, without loss of generality, we can restrict our attention to surjective semantics maps and hence to complete lattices.

3.1.2 Effective Consequence Relations

Usually, due to their non-effective nature, semantic CR's are neither useful for carrying out derivations, nor representable in a machine. They are just declaring "what follows from what." For these reasons, we aim to *represent* a semantic CR by means of another consequence relation, which is defined by *effective* tools.

Definition 3.3 (Representation) *Let* Φ *be a set of formulæ. A* representation *of a CR* $\models$ *on* Φ *is a method for defining a CR* $\vdash$ *on* Φ *which is*

- sound *with respect to* $\models$*, that is* $\vdash \subseteq \models$*;*
- effective*, that is, for* $\Gamma \in \mathcal{P}_{\text{re}}(\Phi)$*,* $A \in \Phi$*, given an evidence representing the fact that* $\Gamma \vdash A$*, then such an evidence can be mechanically checked.*

A representation is complete *with respect to* $\models$ *if it is sound and moreover* $\vdash \supseteq \models$.

These methods are usually defined by means of *proof systems* or *tableaux*. Indeed, a proof system S is nothing but a means for defining a CR which approximates a semantic consequence relation. Such an "effective" CR is seen as "syntactic", and it usually is denoted by a "single turnstile" symbol like "$\vdash_S$".

3.1.3 Noteworthy Examples

We give now some important examples of Consequence Relations. These examples will be useful later.

Consequence Relations for Propositional Logics

Let Φ be a propositional language, and $\Phi_0 \subseteq \Phi$ the set of *atomic propositions* (the variables of Φ). Let Γ range over sets of formulæ. Semantics of Φ is given in terms of *propositional environments*: truth assignments to atomic propositions $PropEnv \stackrel{\text{def}}{=} \Phi_0 \to \{\bot, \top\}$. A propositional environment $\rho \in PropEnv$ is then compositionally extended to formulæ and sets of formulæ, in the usual way.

Despite the simplicity of propositional logic and its interpretation, many consequence relations arise; here we present only those we are interested in. Let $\Gamma \subseteq \Phi$, and $\varphi \in \Phi$, we define

- let $\rho \in PropEnv$; the *(propositional) truth CR with respect to* ρ is

$$\Gamma \models_\rho \varphi \iff (\rho(\Gamma) = \top \Rightarrow \rho(\varphi) = \top)$$

- the *(propositional) truth CR* is $\models \stackrel{\text{def}}{=} \bigcap_{\rho \in PropEnv} \models_\rho$; in other words,

$$\Gamma \models \varphi \iff \forall \rho . \rho(\Gamma) = \top \Rightarrow \rho(\varphi) = \top$$

- let $\rho \in PropEnv$; the *(propositional) validity CR* is

$$\Gamma \Models \varphi \iff (\forall \rho . \rho(\Gamma) = \top) \Rightarrow (\forall \rho . \rho(\varphi) = \top)$$

Other consequence relations arise if we restrict quantifications to subsets of *PropEnv*, instead of taking care of any environment. For instance, in FOL with equality, the atomic predicates "$t_1 = t_2$" have a precise meaning, and cannot be interpreted freely. From a propositional point of view, this correspond to limit the range of quantifications to the environments that correctly interpret "$t_1 = t_2$" as an equality statement.

The propositional truth CR's are at the base of the usual notions of consequence. For instance, a model for a First-Order Logic is composed by the domain of individuals and the evaluation of relational symbols. Therefore, the choice of a precise model corresponds to choose a precise interpretation of atomic formula, which, from a propositional point of view, play the rôle of atomic propositions. First-order logical consequences in a specific model are hence truth propositional consequences with respect to a specific propositional environment. On the other hand, the consequences with respect to the whole class of models are truth consequences with respect to the whole class of propositional environments.

Propositional validity CR's are less common. An example is the *global consequence* for modal logics [vB83, Definition 2.32]. Let Φ a modal language, and F range over frames. A formula φ is *valid in a frame* F *(*$F \models \varphi$*)* if for every assignment to propostional symbols, φ holds in every world of the frame. A formula φ is a *global consequence of* Γ if $\forall F . F \models \Gamma \Rightarrow F \models \varphi$.

Actually, propositional truth and validity CR's can be defined as free consequence relations of suitable semantics of formulæ.

Proposition 3.1 *Let Φ a propositional language, and let ρ range over the set PropEnv of environments for Φ. Then,*

- $\models_\rho$ *is the free CR relative to ρ, in the two-points lattice;*
- $\models$ *is the free CR relative to the semantics (in the lattice $\langle \mathcal{P}(PropEnv), \subseteq \rangle$)*

$$\begin{aligned} [\![\cdot]\!] \quad : \quad & \Phi \to \mathcal{P}(PropEnv) \\ & \varphi \mapsto \{\rho \mid \rho(\varphi) = \top\} \end{aligned}$$

- $\Vdash$ *is the free CR relative to the semantics*

$$\begin{aligned} [\![\cdot]\!] \quad : \quad & \Phi \to \{\bot, \top\} \\ & \varphi \mapsto \bigwedge_{\rho \in PropEnv} \rho(\varphi) \end{aligned}$$

Consequence Relations for First-Order Logics

In a first-order language $\mathcal{L}$, there are two basic syntactic sorts:

- the *terms*, T, ranged over by t, u; a particular subset $Var \subset T$ of atomic terms are the *(object) variables.*
- the *formulæ*, Φ, ranged over by φ, ψ.

Let Γ range over sets of formulæ. Semantics of first order languages is given in terms of *first-order structures.* Let $\mathcal{M} = \langle D, \ldots \rangle$ be a first-order structure for Φ, and $\rho : Var \to D$ an *environment* (a value assignment for variables); then, the *interpretations* of terms and formulæ are two functions $[\![\cdot]\!]_{\mathcal{M}}\rho : T \to D$ and $[\![\cdot]\!]_{\mathcal{M}}\rho : \Phi \to \{\bot, \top\}$, defined compositionally on the syntax. Interpretation of formulæ is extended to sets of formulæ in the obvious way.

From these syntactic and semantic definitions, two main classes of Consequence Relations arise, accordingly to how we intend the meaning of free variables in formulæ. Even more choices arise when we restrict our interpretation to only one model, or a class of structures. We give the following general definitions:

Definition 3.4 *Let $\mathcal{L} = \langle T, \Phi \rangle$ a first-order language and $[\![\cdot]\!]_{\mathcal{M}}$ its interpretation, as above. We define*

Truth CR's: *a.k.a.* local logical consequences*:*

- *the* truth CR with respect to $\mathcal{M}$ *is*

$$\Gamma \models_{\mathcal{M}} \varphi \iff (\forall \rho. [\![\Gamma]\!]_{\mathcal{M}}\rho = \top \Rightarrow [\![\varphi]\!]_{\mathcal{M}}\rho = \top)$$

- *let Λ be a set of first-order structures for L; the* truth CR with respect to Λ *is*

$$\Gamma \models_{\Lambda} \varphi \iff (\forall \mathcal{M} \in \Lambda \forall \rho. [\![\Gamma]\!]_{\mathcal{M}}\rho = \top \Rightarrow [\![\varphi]\!]_{\mathcal{M}}\rho = \top)$$

- *the* truth CR *is* $\models \stackrel{\text{def}}{=} \bigcap_{\mathcal{M}} \models_{\mathcal{M}}$*, where* $\mathcal{M}$ *ranges over all the first-order structures (of the right signature); in other words,*

$$\Gamma \models \varphi \iff \forall \mathcal{M} \forall \rho. [\![\Gamma]\!]_{\mathcal{M}} \rho = \top \Rightarrow [\![\varphi]\!]_{\mathcal{M}} \rho = \top$$

Validity CR's: *a.k.a.* global logical consequences*:*

- *the* validity CR with respect to $\mathcal{M}$ *is*

$$\Gamma \Vdash_{\mathcal{M}} \varphi \iff (\forall \rho. [\![\Gamma]\!]_{\mathcal{M}} \rho = \top) \Rightarrow (\forall \rho. [\![\varphi]\!]_{\mathcal{M}} \rho = \top)$$

- *let* Λ *be a set of first-order structures for* L*; the* validity CR w.r.t. Λ *is*

$$\Gamma \Vdash_{\Lambda} \varphi \iff \forall \mathcal{M} \in \Lambda. (\forall \rho. [\![\Gamma]\!]_{\mathcal{M}} \rho = \top) \Rightarrow (\forall \rho. [\![\varphi]\!]_{\mathcal{M}} \rho = \top)$$

- *the* validity CR *is* $\Vdash \stackrel{\text{def}}{=} \bigcap_{\mathcal{M}} \Vdash_{\mathcal{M}}$*, where* $\mathcal{M}$ *ranges over all the first-order structures (of the right signature); in other words,*

$$\Gamma \Vdash \varphi \iff \forall \mathcal{M} (\forall \rho. [\![\Gamma]\!]_{\mathcal{M}} \rho = \top) \Rightarrow (\forall \rho. [\![\varphi]\!]_{\mathcal{M}} \rho = \top)$$

As an example of interesting class of structures, consider the class PA of structures for Peano Arithmetic, or even the only standard model ($\mathcal{N}$).

We may wonder how to represent any of these abstract CR by means of a proof system. Actually, representations of a truth CR can be very different from those of the corresponding validity CR. For instance, the *generalization* rule $\frac{\varphi}{\forall x \varphi}$ is unsound with respect to $\models$; in order to represent truth CR's we need to add the well-known "freshness condition" on x. Moreover, apparently similar CR's, such as $\models_{PA}$ and $\models_{\mathcal{N}}$, may yield completely different proof systems—indeed, there is *no* finitary proof system complete with respect to $\models_{\mathcal{N}}$, as Gödel proved.

Both truth and validity consequence relations can be defined as free consequence relations of suitable semantics of formulæ.

In truth consequence relations, differently from validity CR's, also assumptions which does not hold only in every environment, are relevant. In fact, we are interested in the set of assignments which enforce a formula to hold. The right semantic space for these CR's is the powerset of the environment space:

Proposition 3.2 *Let* $\mathcal{M} = \langle D, \ldots \rangle$ *a first order structure for* Φ*, and* $Env_{\mathcal{M}} = Var \to D$ *the set of environments on* $\mathcal{M}$*. Consider the lattice* $\langle \mathcal{P}(Env_{\mathcal{M}}), \subseteq \rangle$*, and define* $[\![\varphi]\!]'_{\mathcal{M}} \stackrel{\text{def}}{=} \{\rho \mid [\![\varphi]\!]_{\mathcal{M}} \rho = \top\}$*. Then,* $\models_{\mathcal{M}}$ *is the free consequence relation on* $[\![\cdot]\!]_{\mathcal{M}}$*.*

Let Λ *a set of models; and define* $D_{\Lambda} \stackrel{\text{def}}{=} \langle D, \sqsubseteq \rangle$*, where*

- $D \stackrel{\text{def}}{=} \prod_{\mathcal{M} \in \Lambda} \mathcal{P}(Env_{\mathcal{M}})$*;*
- $d_1 \sqsubseteq d_2 \iff \forall \mathcal{M} \in \Lambda. d_1(\mathcal{M}) \subseteq d_2(\mathcal{M})$*;*
- $(\sqcap A)(\mathcal{M}) \stackrel{\text{def}}{=} \bigcap \{d(\mathcal{M}) \mid d \in A\}$*;*
- $\top(\mathcal{M}) \stackrel{\text{def}}{=} Env_{\mathcal{M}}$*.*

Let $[\![\cdot]\!]'_\Lambda \stackrel{\text{def}}{=} \{\langle \mathcal{M}, [\![\varphi]\!]'_\mathcal{M}\rangle \mid \mathcal{M} \in \Lambda\}$. *Then,* $\models_\Lambda$ *is the free consequence relation on* $[\![\cdot]\!]'_\Lambda$. *In particular, for* $\Lambda =$*the set of all first-order structures, the free consequence relation on* $[\![\cdot]\!]'_\Lambda$ *is* $\models$.

On the other hand, validity CR's have an "all or nothing" flavour. A formula either is a theorem (it holds in every environment), or it is not. We do not take care of differences among formulæ which are not theorems: from the validity point of view, "$x = 0$" and "*false*" mean the same. The right semantic space for these CR's is the two-points lattice:

Proposition 3.3 *Let* $\mathcal{M} = \langle D, \ldots\rangle$ *a first order structure for* Φ, *and* $\mathrm{Env}_\mathcal{M} = \mathrm{Var} \to D$ *the set of environments on* $\mathcal{M}$. *Consider the 2-point lattice* $\langle\{\bot, \top\}, \sqcap\rangle$, *and define* $[\![\cdot]\!]''_\mathcal{M} \stackrel{\text{def}}{=} \begin{cases} \top & \text{if } \forall\rho.[\![\varphi]\!]_\mathcal{M}\rho = \top \\ \bot & \text{otherwise.} \end{cases}$ *Then,* $\Vdash_\mathcal{M}$ *is the free consequence relation on* $[\![\cdot]\!]''_\mathcal{M}$.

Let Λ *a set of models; and define the lattice* $D_\Lambda \stackrel{\text{def}}{=} \langle D, \sqsubseteq\rangle$, *where*

- $D \stackrel{\text{def}}{=} \Lambda \to \{\bot, \top\}$;
- $d_1 \sqsubseteq d_2 \iff \forall\mathcal{M} \in \Lambda. d_1(\mathcal{M}) \leq d_2(\mathcal{M})$;
- $(\sqcap A)(\mathcal{M}) \stackrel{\text{def}}{=} \sqcap\{d(\mathcal{M}) \mid d \in A\}$;
- $\top(\mathcal{M}) \stackrel{\text{def}}{=} \top$.

Let $[\![\cdot]\!]''_\Lambda = \{\langle \mathcal{M}, [\![\varphi]\!]''_\mathcal{M}\rangle \mid \mathcal{M} \in \Lambda\}$. *Then,* $\Vdash_\Lambda$ *is the free consequence relation on* $[\![\cdot]\!]''_\Lambda$. *In particular, for* $\Lambda =$*the set of all first-order structures, the free consequence relation on* $[\![\cdot]\!]''_\Lambda$ *is* $\Vdash$.

Consequence Relations for Propositional Modal Logics

In *modal logics*, two further kinds of CR's arise, according to whether we consider consequence with respect to *worlds* or *frames*.

Let Φ a propositional modal languages, such as $\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \psi \mid \Box\varphi$, where p ranges over propositional variables Φ_0. In terms of arities, this language is defined by the following alphabet:

$$\begin{array}{rcl} \mathcal{S} = \mathcal{V} & = & \{\Phi\} \\ \mathcal{E} & = & \{\neg^{\Phi\to\Phi}, \wedge^{(\Phi,\Phi)\to\Phi}, \Box^{\Phi\to\Phi}\} \end{array}$$

Semantics of modal logics is given on *Kripke structures* [HC84, vB83]. Let F range over *frames*, that is pairs $\langle S, R\rangle$ where S is a sets (the *states*), and $R \subseteq S \times S$ is a relation (the *accessibility relation*) between states. Let ρ range over environments (functions $\Phi_0 \to \mathcal{P}(S)$). A *model* is a pair $\mathcal{M} = \langle F, \rho\rangle$; the interpretation of a formula φ in the model $\langle F, \rho\rangle$ is a set of states $[\![\varphi]\!]_F\rho \subseteq F$, defined by induction on the syntax of φ. Interpretations are extended to subsets of Φ:

$$[\![p]\!]_F\rho = \rho(p) \qquad [\![\neg\varphi]\!]_F\rho = S \setminus [\![\varphi]\!]_F\rho \qquad [\![\varphi \wedge \psi]\!]_F\rho = [\![\varphi]\!]_F\rho \cap [\![\psi]\!]_F\rho\}$$
$$[\![\Box\varphi]\!]_F\rho = \{s \in S \mid \forall s' \in S. R(s, s') \Rightarrow s' \in [\![\varphi]\!]_F\rho\} \qquad [\![\Gamma]\!]_F\rho = \bigcup_{\varphi\in\Gamma} [\![\varphi]\!]_F\rho$$

Similarly to what happens with FOL, there are two main classes of Consequence Relation, *truth* and *validity* CR's, accordingly to whether we require formulæ to hold in every state or not. (This is not surprising, if we recall how modal logics can be translated in first order logics.) Even more choices arises when we restrict our interpretation to only one model, or a class of structures. We give the following general definitions:

Truth CR's: a.k.a. *model local consequences* [vB83]:

- the *truth CR with respect to* $\mathcal{M} = \langle F, \rho \rangle$ is

$$\Gamma \models_{\mathcal{M}} \varphi \iff \forall w. w \in [\![\Gamma]\!]_F \rho \Rightarrow w \in [\![\varphi]\!]_F \rho$$

- let Λ be a set of models; the *truth CR with respect to* Λ is

$$\Gamma \models_{\Lambda} \varphi \iff \forall \mathcal{M} \in \Lambda. \Gamma \models_{\mathcal{M}} \varphi$$

- the *truth CR* is $\models \stackrel{\text{def}}{=} \bigcap_{\mathcal{M}} \models_{\mathcal{M}}$, where $\mathcal{M}$ ranges over all modal models.

Validity CR's: a.k.a. *model global consequences* [vB83]:

- the *validity CR with respect to* $\mathcal{M} = \langle F, \rho \rangle$ is

$$\Gamma \Vdash_{\mathcal{M}} \varphi \iff [\![\Gamma]\!]_F \rho = S \Rightarrow [\![\varphi]\!]_F \rho = S$$

- let Λ be a set of models; the *validity CR with respect to* Λ is

$$\Gamma \Vdash_{\Lambda} \varphi \iff \forall \mathcal{M} \in \Lambda. \Gamma \Vdash_{\mathcal{M}} \varphi$$

- the *validity CR* is $\Vdash \stackrel{\text{def}}{=} \bigcap_{\mathcal{M}} \Vdash_{\mathcal{M}}$, where $\mathcal{M}$ ranges over all modal models.

As an example of interesting class of structures, consider the set of frames whose accessibility relation is transitive, or reflexive, or both...

As for previous logics, both truth and validity consequence relations for modal logics can be defined as free consequence relations of suitable semantics of formulæ. The same considerations of first-order logics apply, *mutatis mutandis*.

Proposition 3.4 *Let $F = \langle S, R \rangle$ a frame, $\mathcal{M} = \langle F, \rho \rangle$ a modal model for Φ, and $[\![\cdot]\!]_{\mathcal{M}} \stackrel{\text{def}}{=} [\![\cdot]\!]_F \rho : \Phi \to \mathcal{P}(S)$ the semantics of formulæ, defined as above. Then $\models_{\mathcal{M}}$ is the free consequence relation on $[\![\cdot]\!]_{\mathcal{M}} : \Phi \to \mathcal{P}(S)$.*

Let Λ a set of models; and define $D_\Lambda \stackrel{\text{def}}{=} \langle D, \subseteq, \bigcap, \top \rangle$, where

- $D \stackrel{\text{def}}{=} \prod_{\mathcal{M} \in \Lambda} \mathcal{P}(S_{\mathcal{M}})$;
- $d_1 \subseteq d_2 \iff \forall \mathcal{M} \in \Lambda. d_1(\mathcal{M}) \subseteq d_2(\mathcal{M})$;
- $(\bigcap A)(\mathcal{M}) \stackrel{\text{def}}{=} \bigcap \{d(\mathcal{M}) \mid d \in A\}$;
- $\top(\mathcal{M}) \stackrel{\text{def}}{=} S_{\mathcal{M}}$.

Let $[\![\cdot]\!]_\Lambda = \{\langle \mathcal{M}, [\![\varphi]\!]_\mathcal{M}\rangle \mid \mathcal{M} \in \Lambda\}$. *Then,* $\models_\Lambda$ *is the free consequence relation on* $[\![\cdot]\!]_\Lambda$. *In particular, for* $\Lambda =$*the set of all models, the free consequence relation on* $[\![\cdot]\!]_\Lambda$ *is* $\models$.

Proposition 3.5 *Let* $F = \langle S, R\rangle$ *a frame and* $\mathcal{M} = \langle F, \rho\rangle$ *a modal model for* Φ, *and* $[\![\cdot]\!]_\mathcal{M} \stackrel{\text{def}}{=} [\![\cdot]\!]_F\rho : \Phi \to \mathcal{P}(S)$ *the semantics of formulæ, defined as above. Consider the 2-points lattice* $\langle\{\bot, \top\}, \sqcap\rangle$, *and define* $[\![\cdot]\!]'_\mathcal{M} \stackrel{\text{def}}{=} \begin{cases} \top & \text{if } [\![\varphi]\!]_\mathcal{M} = S \\ \bot & \text{otherwise.} \end{cases}$ *Then,* $\Vdash_\mathcal{M}$ *is the free consequence relation on* $[\![\cdot]\!]'_\mathcal{M}$.

Let Λ *a set of models; and define* $D_\Lambda \stackrel{\text{def}}{=} \langle D, \sqsubseteq, \sqcap, \top\rangle$, *where*

- $D \stackrel{\text{def}}{=} \Lambda \to \{\bot, \top\}$;
- $d_1 \sqsubseteq d_2 \iff \forall \mathcal{M} \in \Lambda. d_1(\mathcal{M}) \sqsubseteq d_2(\mathcal{M})$;
- $(\sqcap A)(\mathcal{M}) \stackrel{\text{def}}{=} \sqcap\{d(\mathcal{M}) \mid d \in A\}$;
- $\top(\mathcal{M}) \stackrel{\text{def}}{=} \top$.

Let $[\![\cdot]\!]'_\Lambda = \{\langle \mathcal{M}, [\![\varphi]\!]'_\mathcal{M}\rangle \mid \mathcal{M} \in \Lambda\}$. *Then,* $\Vdash_\Lambda$ *is the free consequence relation on* $[\![\cdot]\!]'_\Lambda$. *In particular, for* $\Lambda =$*the set of all models, the free consequence relation on* $[\![\cdot]\!]'_\Lambda$ *is* $\Vdash$.

Further consequence relation will be introduced later, in Section 3.2.3, and in Part II.

3.2 Schematic Consequence Relations

Logics are usually described by using axiom and rule *schemata*; these schemata can be *instantiated* in order to get closed (i.e. "particular") axioms and inference steps. Moreover, instantiations may themselves be schematic: a derivation composed by schematic rules may be seen as a *derived* rule (schemata), which can be further instantiated and applied.

3.2.1 Deterministic and Non-Deterministic Substitutions

We need a syntax-free treatment of instantiation and schematicity. We start from a quite general approach to substitution, such as Aczel's [Acz94]:

Definition 3.5 (Deterministic Substitutions) *Let* Φ *be a set of formulæ, and* $Sub \subseteq \Phi^\Phi$. *Sub is a set of* deterministic substitutions *on* Φ *if it forms a submonoid of* $\langle \Phi^\Phi, id_\Phi, \circ\rangle$. *The pair* $\langle \Phi, Sub\rangle$ *is called a* concrete monoid; *the set of concrete monoids is denoted by CM. For* $A \in \Phi$, $\sigma \in Sub$, $A\sigma$ *is the* instance *of* A *under* σ.

In this setting we do not commit ourselves to any specific kind of syntax or substitution. Definition 3.5 is general enough to capture a wide range of usual substitutions (e.g. those of propositional logics, first-order logics, λ-calculus, Hoare Logic,...).

Defining substitutions as functions is a common solution, but it is not always adequate. Indeed, a substitution function has a deterministic flavour, while, on the contrary, in many languages and logics substitutions are non-deterministic. For instance, the usual substitution "$\varphi[t/x]$" of FOL is nondeterministic, because we require to "rename bound variables in order to avoid capture," but we do not specify explicitly *how* this renaming

has to be carried out. Hence, by substituting x for y in $\forall x.y$ we can obtain both $\forall z.x$ and $\forall w.x$. Of course, each of these formulæ can be seen as the result of applying a specific substitution function, say $[x/y]_z$ and $[x/y]_w$ respectively, but in general we do not take care of these details because they are not relevant at the logical level.

Therefore, we introduce a more general definition of substitution, by defining them as *relations* between formulæ:

Definition 3.6 (Substitutions Relations and Schemata) *Let Φ be a set of formulæ. A set $Sub \subseteq \mathcal{P}(\Phi \times \Phi)$ is a* substitution schema (on Φ) *if*

- *it contains the* identity substitution $id_\Phi \stackrel{\text{def}}{=} \{\langle A, A\rangle \mid A \in \Phi\} \in Sub$
- *it is closed under relation composition:* $\forall \sigma_1, \sigma_2 \in Sub : (\sigma_1 \circ \sigma_2) \in Sub$

We denote by SS the set of pairs $\langle \Phi, Sub\rangle$.

Each element of a substitution schema is a substitution relation *(or simply a* substitution*).*

For $A \in \Phi$, $\sigma \in Sub$, B is the instance *of A under σ (written $B \in A\sigma$) if $A\sigma B$.*

A substitution is total *iff $\forall \sigma \in Sub. \forall A \in \Phi : \exists B \in \Phi.(A\sigma B)$.*

A substitution is deterministic *iff $\forall A \exists! B. A\sigma B$. We denote by $detSS \subset SS$ the substitution schemata of only deterministic substitution relations.*

Substitutions on formulæ can be naturally extended to sets: $\forall \sigma \in Sub, \forall \Gamma_1, \Gamma_2 \in \mathcal{P}(\Phi)$, we define $\Gamma_2 \in \Gamma_1\sigma \iff \forall B \in \Gamma_2 \exists A \in \Gamma_1 : B \in A\sigma$

Notice that each substitution schema is a submonoid of $\langle \mathcal{P}(\Phi \times \Phi), id_\Phi, \circ\rangle$, where $\circ$ is the usual composition of relations. There is an obvious isomorphism between concrete monoids and monoids of deterministic substitution relations:

Proposition 3.6 *The function*

$$\begin{array}{rccl} f: & CM & \to & detSS \\ & \langle \Phi, Sub\rangle & \mapsto & \langle \Phi, \{\{\langle A, \sigma(A)\rangle \mid A \in \Phi\} \mid \sigma \in Sub\}\rangle \end{array}$$

is a monoid isomorphism.

Notation remark. In the following, for sake of simplicity we will write substitution relations in a functional fashion, possibly by composing and nesting inside formulæ. Therefore, by $(A\sigma)\delta$ we denote the set of formulæ $\{B \mid \exists C \in A\sigma. B \in C\delta\}$. We will still denote, however, the *set* of every possible instance. For instance, in first-order logic we will write "$\varphi[t/x] \wedge \psi[t/x]$" for the set $\{\varphi' \wedge \psi' \mid \varphi' \in \varphi[t/x], \psi' \in \psi[t/x]\}$. (In the case of deterministic substitution schemata, these sets are singletons.)

3.2.2 (Free) Schematic Consequence Relations

We proceed now to formalize the notion of schematicity in consequence relations. A schematic CR is then simply a CR closed under substitution:

Definition 3.7 (Schematic Consequence Relation) *Let Φ be a set of formulæ, and Sub a substitution schema on Φ. A consequence relation $\rhd$ on Φ is* schematic *(relatively to Sub) if it is closed under Sub, i.e. if $\Gamma \rhd A$ then $\forall \sigma \in Sub$, $\forall \Gamma' \in \Gamma\sigma$, $\forall A' \in A\sigma : \Gamma' \rhd A'$.*

An interesting case arises when the consequence relation is defined by means of a truth semantics, via the free construction of Definition 3.2. In this case, schematicity of a free CR can be reduced to the "monotonicity" of the semantics of substitutions relations with respect to the order of the semantic lattice.

Definition 3.8 *Let $[\![\cdot]\!] : \Phi \to D$ be a semantics for Φ, and Sub a substitution schema for Φ.*

For $\sigma \in Sub$, the semantics of σ *is the least relation $[\![\sigma]\!] \subseteq D \times D$ such that $\forall \varphi, \psi \in \Phi : [\![\varphi]\!][\![\sigma]\!][\![\psi]\!] \iff \varphi\sigma\psi$.*

The semantics of *Sub is the set of relations $[\![Sub]\!] \stackrel{\text{def}}{=} \{[\![\sigma]\!] \mid \sigma \in Sub\} \subseteq D \times D$.*

Proposition 3.7 *Let Φ be a set of formulæ and Sub a substitution schemata on Φ. Let $\langle D, \sqsubseteq \rangle$ be a lattice, and $[\![\cdot]\!] : \Phi \to D$ be a semantics of Φ in D. Then, the following are equivalent:*

1. *$\forall s \in [\![Sub]\!] \forall d_1, d_2, e_1, e_2 \in D$, if $d_1 s e_1$, $d_2 s e_2$ and $d_1 \sqsubseteq e_1$, then $d_2 \sqsubseteq d_2$;*
2. *the free CR $\models$ on $[\![\cdot]\!]$ is schematic relatively to Sub.*

Proof. **1 ⇒ 2**. Let $\Gamma \models \varphi$ and $\sigma \in Sub$, and let $\Gamma' \in \Gamma\sigma, \varphi' \in \varphi\sigma$; we have to prove $\Gamma' \models \varphi'$. By definition of $[\![\sigma]\!]$, $[\![\Gamma]\!][\![\sigma]\!][\![\Gamma']\!]$ and $[\![\varphi]\!][\![\sigma]\!][\![\varphi']\!]$. By hypothesis 1, we have $[\![\Gamma']\!] \subseteq [\![\varphi']\!]$, hence the thesis.

2 ⇒ 1. Let $\models$ be schematic relatively to *Sub* and $\sigma \in Sub$, and let $d_1 [\![\sigma]\!] d_2$ and $e_1 [\![\sigma]\!] e_2$. By definition of $[\![\sigma]\!]$, there are $\varphi_1, \varphi_2, \psi_1, \psi_2 \in \Phi$ such that $[\![\varphi_i]\!] = d_i, [\![\psi_i]\!] = e_i$ for $i = 1, 2$ (outside the image of $[\![\Phi]\!]$, $[\![\sigma]\!]$ is undefined), and $\varphi_1 \sigma \varphi_2$, $\psi_1 \sigma \psi_2$. Suppose $d_1 \sqsubseteq e_1$; then, $\varphi_1 \models \psi_1$ and by schematicity of $\models$ we have $\varphi_2 \models \psi_2$, that is $d_2 \sqsubseteq e_2$. □

Condition (1) above is just a kind of "relational monotonicity." Indeed, if we consider deterministic substitution schemata, the condition can be restated as "$\forall s \in [\![Sub]\!] \forall d, e : d \sqsubseteq e \Rightarrow s(d) \sqsubseteq s(e)$," which is nothing but monotonicity of substitution functions.

3.2.3 Noteworthy examples

Most known consequence relations are schematic, in the appropriate sense. In this section we examine two cases of interest. The first concerns First-Order Logics, where we deal with the standard notions of schematicity and substitution—this case is particularly relevant, since most logics feature substitution notions which behave similarly to those of FOL.

In the second case, we deal with First-Order Dynamic Logic. The standard substitution notions fail for this logic, due to the presence of commands. Nevertheless, we will see how is it still possible to define a suitable substitution schema.

Schematicity of First-Order Logics

In First-Order Logic, we usually refer to three notions of substitution:

1. substitution of formulæ for propositional variables in formulæ. Actually, this substitution notion regards the "propositional level" of the logic, and it is common to every logic in which we speak of "propositional variables" and axiom and rule

schemata (see, e.g., the Modus Ponens.) We represent this notion by a deterministic substitution schema Sub_p on Φ, which is the least set closed by composition such that, for each $p \in \Phi_0$, $\varphi \in \Phi$, it contains the substitution function $[\varphi/p] : \Phi \to \Phi$, whose definition is as usual.

2. substitution of terms for object variables in terms; it is represented by a deterministic substitution schema Sub_i on T, which is the least set closed by composition such that for each $x \in Var$, $t \in T$, it contains the substitution function $[t/x] : T \to T$, whose definition is the usual;

3. substitution of terms for object variables in formulæ; it is represented by a nondeterministic substitution schema Sub_o on Φ, which is the least set closed by composition such that for each $x \in Var$, $t \in T$, it contains the simple substitution relation $[t/x] \in \mathcal{P}(\Phi \times \Phi)$, defined on the syntax as follows (we list only some cases):

$$\begin{aligned}
(R(t_1, \ldots, t_n))[t/x] &= R(t_1[t/x], \ldots, t_n[t/x]) \\
(\varphi \wedge \psi)[t/x] &= \varphi[t/x] \wedge \psi[t/x] \\
(\forall x\varphi)[t/x] &= \{\forall x\varphi\} \\
(\forall y\varphi)[t/x] &= \{\forall z.\psi \mid z \notin \mathrm{FV}(\varphi, t), \psi \in (\varphi[z/y])[t/x]\} \quad \text{for } x \neq y
\end{aligned}$$

where R ranges over relation symbols of the logic.

It is evident that term substitution on formulæ is highly non deterministic: just recall that, as remarked earlier, substitutions are treated as nondeterministic functions.

Proposition 3.8 (Schematicity of CR's for FOL) *Let $\langle T, \Phi\rangle$ a FOL; then*

1. *both truth and validity consequence relations are schematic with respect to Sub_p.*
2. *the truth consequence relations are schematic with respect to Sub_o;*
3. *If there is a model $\mathcal{M}$ and $\varphi \in \Phi$, $t, x \in T$ such that*

$$\emptyset \neq [\![\varphi]\!]'_{\mathcal{M}} \neq [\![\varphi[t/x]]\!]'_{\mathcal{M}} = \mathcal{P}(\mathrm{Env}_{\mathcal{M}}),$$

then the free CR $\models_{\mathcal{M}}$ on $[\![\cdot]\!]''_{\mathcal{M}}$ is not schematic with respect to Sub_o.

Proof. **1.** We sketch the case for truth CR (the case for validity CR is the same). Let $\Gamma \models_{\mathcal{M}} \varphi$, $p \in \Phi_0$, $\psi \in \Phi$, and $e : \Phi_0 \to \{\bot, \top\}$ a truth assignment to propositional variables. Then, define

$$e'(q) \stackrel{\text{def}}{=} \begin{cases} e(\psi) & \text{if } q \equiv p \\ e(q) & \text{otherwise} \end{cases}$$

By induction on the syntax, it is easy to prove that $\forall \varphi \in \Phi : e(\varphi[\psi/p]) = e'(\varphi)$. Hence, $[\![\Gamma[\psi/p]]\!]_{\mathcal{M}} e = [\![\Gamma]\!]_{\mathcal{M}} e'$ and $[\![\varphi[\psi/p]]\!]_{\mathcal{M}} e = [\![\varphi]\!]_{\mathcal{M}} e'$. Therefore, if $[\![\Gamma[\psi/p]]\!]_{\mathcal{M}} e = T$ then $[\![\Gamma]\!]_{\mathcal{M}} e' = T$, so $[\![\varphi]\!]_{\mathcal{M}} e' = [\![\varphi[\psi/p]]\!]_{\mathcal{M}} e = T$.

2. We see the case of $\models_{\mathcal{M}}$; the others follow trivially. Let $\mathcal{M}$ be a f.o. model, and $\Gamma \models_{\mathcal{M}} \varphi$, $x \in Var$, $t \in T$, and $\rho : Var \to D$ an environment in the model $\mathcal{M}$. Then, define

$$\rho'(y) \stackrel{\text{def}}{=} \begin{cases} \rho(t) & \text{if } x \equiv y \\ \rho(y) & \text{otherwise} \end{cases}$$

By induction on the syntax, it is easy to prove that $\forall \varphi \in \Phi : \rho(\varphi[t/x]) = \rho'(\varphi)$. Hence, $[\![\Gamma[t/x]]\!]_{\mathcal{M}}\rho = [\![\Gamma]\!]_{\mathcal{M}}\rho'$ and $[\![\varphi[t/x]]\!]_{\mathcal{M}}\rho = [\![\varphi]\!]_{\mathcal{M}}\rho'$. Therefore, if $[\![\Gamma[t/x]]\!]_{\mathcal{M}}\rho = T$ then $[\![\Gamma]\!]_{\mathcal{M}}\rho' = T$, so $[\![\varphi]\!]_{\mathcal{M}}e' = [\![\varphi[\psi/p]]\!]_{\mathcal{M}}e = T$; we conclude therefore that $\Gamma[t/x] \models_{\mathcal{M}} \varphi[t/x]$.

3. Since $[\![\varphi]\!]'_M \neq \mathcal{P}(Env_{\mathcal{M}})$, there is $\rho \in Env_{\mathcal{M}}$ such that $[\![\varphi]\!]_{\mathcal{M}}\rho = \bot$. Therefore, $[\![\varphi]\!]''_{\mathcal{M}} = [\![\forall x.\varphi]\!] = \bot$ and hence $\varphi \models_{\mathcal{M}} \forall x.\varphi$ holds.

On the other hand, $\varphi[t/x]$ holds in every environment; therefore $[\![\varphi[t/x]]\!]''_{\mathcal{M}} = \top$, and hence $\varphi[t/x] \not\models_{\mathcal{M}} \forall x\varphi$. Since $(\forall x\varphi)[t/x] = \forall x\varphi$, we have the thesis. □

An important consequence of the last statement is the following result:

Corollary 3.9 *if the logic has the equality, then the validity consequence relations are not schematic with respect to Sub_o.*

Proof. Take $\varphi = (x = y)$ and a model with at least two elements. We can then define an environment ρ such that $\rho(x) \neq \rho(y)$; therefore, $\rho \notin [\![x = y]\!]'$. On the other hand, $(x = y)[y/x]$ is $y = y$, which holds in every environment. By applying the Proposition, therefore, we obtain the thesis. □

Schematicity of Dynamic Logic

There are notions of substitution much more complex than usual ones. An interesting example is that given by first-order Dynamic Logic [Har79, Har84, KT90].

The language of formulæ of First-Order Dynamic Logic is a first-order language extended by adding a modal constructor [Har84, KT90]:

$$\Phi \quad = \quad \varphi ::= \ldots \mid \forall x\varphi \mid [c]\varphi$$

where c ranges over the language of *regular programs:*

$$C \quad = \quad c ::= x := t \mid c_1; c_2 \mid b? \mid c_1 + c_2 \mid c^*$$

where b ranges over the program- and quantifier-free formulæ. (Syntax and semantics of Dynamic Logic will be described in detail in Sections 6.1, 7.1. [2])

The problematic substitution arises in relation with the axiom $\forall x\varphi \supset \varphi[t/x]$ when a program appears in φ. In this case, we cannot apply directly the substitution, since for instance, $\forall x.[x := 0]x = 0$ would yield $[1 := 0]1 = 0$ which is clearly meaningless.

In most systems for Dynamic Logic, this drawback is overcome by replacing every subformula $[c]\varphi$, where x occurs on the left-hand sides of some assignment in c, by an equivalent "special form", $[z := x; c[z/x]; x := z]\varphi$, where z is fresh [Har79, Har84, KT90]. In these special forms, only the first occurrence of x is considered free, and hence instantiated by a substitution. Therefore, a substitution in Dynamic Logic replaces commands by their equivalent special forms, wherever required, before performing instantiation.

Let us formalize this notion. Firstly, we need to deal with the substitution of terms for object variables in commands. Let $\mathbb{AV}(c)$ the set of *assigned variables* of c, that is, the

[2] Briefly, the informal meaning of some of these constructs is the following: $b?$ ="test b; proceed if true, fail if false"; c^* ="execute c a nondeterministically chosen finite number of times"; $[c]\,\varphi$ ="at the end of every non-diverging execution of c, φ holds"; $\langle c\rangle\,\varphi$ ="there is an execution of c such that at its end φ holds".

set of variables which appear on the left-hand side of assignments in c. We cannot replace assigned variables by generic terms, but we can replace a variable by another variable, and a non-assigned variable by a term. Hence, we introduce two substitution schemata on C.

The first one, Sub_{cv}, is the set of substitutions of variables for variables. These are of the form "$c[y/x]_v$", which replaces *every* occurrence of x in c (also those on l.h.s. of assignments) by y. This substitution schema is deterministic.

The second one, Sub_{ct}, is the set of substitutions of terms for non-assigned variables. These substitutions are of the form "$c[t/x]_t$", compositionally defined as follows:

$$(y := t_1)[t/x]_t = y := (t_1[t/x]_t) \qquad (c_1 \, op \, c_2)[t/x]_t = (c_1[t/x]_t) \, op \, (c_2[t/x]_t)$$
$$(b?)[t/x]_t = (b[t/x]_t)? \qquad (c^*)[t/x]_t = (c[t/x]_t)^*$$

where $op \in \{+, ;\}$. Again, this schema is deterministic.

Notice, however, that care has to be taken in applying substitutions of this schema: in general, they do not preserve the meaning of programs. Consider, for instance, $(x := x+1; x := x+1)[0/x]_t$, which is $(x := 0+1; x := 0+1)$, although the intended meanings of these commands are not the same. However, these substitutions are applied only after a suitable renaming of assigned variables. In fact, we have to extend the substitution schema on formulæ we have defined for First-Order Logic (see above). As before substitution of terms for object variables in formulæ is represented by a nondeterministic substitution schema Sub_o on Φ, which is the least set closed by composition such that, for each $x \in Var$, $t \in T$, it contains the simple substitution relation $[t/x] \in \mathcal{P}(\Phi \times \Phi)$. The definition of these "atomic substitutions" is just as the one of FOL, plus the following case for the new constructor:

$$\begin{array}{rcl} \ldots & & \text{(the cases of FOL) } \ldots \\ ([c]\varphi)[t/x] & = & \begin{cases} [c[t/x]_t]\,(\varphi[t/x]) & \text{if } x \notin \mathbb{AV}(c) \\ \{[z := t; c[z/x]_v; x := z]\varphi \mid z \notin \mathbb{FV}(\varphi, c)\} & \text{if } x \in \mathbb{AV}(c) \end{cases} \end{array}$$

Due to the case analysis on the occurrence of variables, this substitution schema preserves the meaning of formulæ, although $[t/x]_t$ does not.

Although this substitution notion is quite different from the "standard" ones, Sub is a good substitution schema. There are two sources of nondetermism: α-conversion for bound variables (like in FOL), and "command conversion" for assigned variables.

We list some examples:

$$\begin{array}{rcl} \varphi[t/x] & \equiv & \text{as usual, if } \varphi \text{ is program free} \\ ([z := t_1]\varphi)[t/x] & \equiv & [z := (t_1[t/x])](\varphi[t/x]) \text{ for } x \neq z \\ (x + 1 = y \supset [x := x + 1]x = y)[t/x] & \equiv & (t + 1 = y \supset [z := t; z := z + 1; x := z]x = y) \end{array}$$

Truth and validity consequence relations for Dynamic Logic can be defined by following Definition 3.4 (see Definition 7.1). By an argument similar to the one of Proposition 3.8, it is easy to prove that the truth CR for Dynamic Logic is schematic with respect to Sub, while the validity one is not.

Remark. As we have already said, this treatment of substitution is adopted in most of the proof systems for DL in the literature. This is acceptable because these proof systems

are Hilbert-style, and they are not supposed to be used interactively in certifying programs. However, this substitution schema becomes cumbersome and hard to understand as soon as we aim to a more user-friendly interactive system for DL, possibly in Natural Deduction style. In fact, replacing programs with equivalent "special forms" yields rapidly weird and obscure formulæ. In Chapter 7 we propose another solution, easier to understand and to use, which takes full advantage of the Natural Deduction style.

3.3 Heterogeneous Consequences Relations

As we noticed before (Section 3.1),more than just one consequence relation arises for any given logic. In FOL, for instance, we have the *validity* consequence relation and the *truth* consequence relation, according to whether we consider free variables in formulæ. In *modal logics* two further kinds of CR's arise, according to whether we consider consequence with respect to *worlds* or with respect to *frames*. Moreover, many more CR's can be defined by restricting attention to interesting subclasses of models. Similarly to what happens for FOL, both validity and truth CR's arise in program logics.

Usually, in defining a proof system we restrict our attention to one particular consequence relation, disregarding all the others. In FOL, for instance, a Hilbert-style system for validity CR is unsound for the truth CR (it features, for instance, the generalization rule $\frac{\varphi}{\forall x \varphi}$), and a ND-style system for truth CR is not complete with respect to the validity CR (it does not derive the generalization rule). In other words, a simple CR deals with one consequence notion at a time.

This approach corresponds to focus on a single *judgement* (in the sense of Martin-Löf [Mar85]) as the main type of assertions expressed by the proof system and hence embodied the consequence relation. Classically, sequents are built up of multisets of formulæ, although what we focus on is *the meaning* of these formulæ. For instance, in First Order Logic we have (at least) two judgements on the same language and semantic structures: φ *is valid* and φ *is true* (with respect to a given model and evaluation.) Many more judgements can be defined on the same language: falsity, satisfability, and so on. Moreover, in most proof systems, beside the main judgement (say, "provability"), there are other judgements on formulæ (or other syntactic objects), whose formalization is often neglected, if not "swept under the rug." Common examples of these judgements in Logics and Computer Science include (but are not limited to)

- the *eigenvariable conditions* of quantifier rules (e.g. in Gentzen's and Prawitz' systems);
- the *boxed assumptions* conditions of Prawitz' systems for modal logics;
- non-interference judgements (e.g. in Hoare Logic and Hoare Logic for concurrency [HM93, AO91]);
- conversions (e.g. in λ-calculi);
- typing and subtyping judgements;
- cells allocation (e.g. in operational and axiomatic semantics of store-base languages [Plo81, HMST93]);
- occurrence checks in model checking [SW89].

In the traditional presentations approach, these notions are seldom formalized by means of specific proof systems, on a par with the one for the "main" judgement. Even less common are systems in which several "logical" consequences are formalized simultaneously (e.g., systems which deal with both truth and validity of formulæ).

Therefore, it is important to study how to handle more than one judgement at the same time, and not only more than one "logical" judgements, but also those which are usually regarded as "auxiliary." This approach also naturally arises from the ability of Logical Frameworks of treating more than one judgement at the same time. This can be used, for instance, in order to treat simultaneously together several logical notions, or to enforce side conditions such as the forementioned ones.

In the traditional presentations approach, CR's are *homogeneous*, that is every formulæ occurring in (multi)sets take meaning in the *same* domain. However, it is useful also to consider non homogeneous (multi)sets of formulæ, where this does not happen (Notice that formulæ may be of the same language and still being interpreted in different ways). Examples of heterogeneous systems arise from the above systems when we spell out the formal conditions for side conditions: side conditions are nothing but a different kind of formulæ, whose interpretation is not the same as that of the formulæ the system focuses on. A semantic version of non-homogeneus CR are mixed truth-validity single-conclusioned consequence relation, such as the following:

$$\begin{array}{rcl} \rhd & \subset & (\mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)) \times \Sigma \\ \Gamma; \Delta \rhd \varphi & \iff & \forall M.(\forall \rho.\rho \models \Gamma) \Rightarrow (\forall \rho.\rho \models \Delta \Rightarrow \rho \models \varphi) \end{array}$$

A similar relation will been introduced for Modal Logics in Chapter 5 (see also [AHMP97]).

In order to formalize heterogeneous consequence relations, the notion of simple CR can be generalized in several ways.

3.3.1 Judgement Consequence Relations

A first approach is to put another layer between syntactic formulæ and Consequence Relations. Since we focus on is *the meaning* of formulæ, it is natural to take *judgements* as components of multisets, instead of simple formulæ. This approach has been exploited by Martin-Löf in his Theory of Judgements [Mar85]. Assumptions and conclusions of consequence relations are no more formulæ, such as "φ", but assertions such as "φ *true*", "x *not free in* φ", and so on. These assertions form the *basic* (or *atomic*) *judgements* and characterize the logic. Apart from the basic judgements, we can build other judgements, called *higher-order judgements*, by combining basic (and derived) judgements in the following two ways:

- if J_1 and J_2 are two judgements, then $J_1 \to J_2$ is an *hypothetical judgement*. It expresses a notion of "consequence" of J_2 from J_1: "J_2 is derivable (or provable, or a consequence) of J_1";
- if J is a judgement involving a variable x ranging over a sort S, then $\bigwedge_{x \in S} J$ is an *schematic* (or *general*) *judgement*. It expresses a notion of "generality" of J with respect to S: "J is derivable (or provable, or holds) for every x in C".

It is worthwhile noticing that these are *metalogical* constructions, and *not* formula constructors. For instance, a hypothetical judgement (φ *true*) $\to$ (ψ *true*) should be read

"if we assume φ, then we prove ψ". This is completely different from the meaning of the basic judgement $(\varphi \supset \psi)$ *true*, which means "the formula $(\varphi \supset \psi)$ is provable".

The encapsulation of formulæ inside judgements let us to consider uniformly heterogeneous formulæ. For instance, both "φ *true*," "φ *valid*" and "x *not free in* φ" are three basic judgements. It is therefore natural to consider simple consequence relations of basic judgements, instead of consequence relations of formulæ.

In the following, by "a set of indexes" we mean an initial segment of $\mathcal{N}$, that is an element I of the set $\{\{i \mid 1 \leq i \leq n\} \mid n \in \mathcal{N}\}$ (notice that these sets are recursive). We give then the following definition:

Definition 3.9 (Judgement Consequence Relation) *Let I be a set of indexes, $(\Phi_i)_i$ a family of sets of formulæ, for $i \in I$, (possibly with repetitions).*

The assertions for *I are the elements of the set $\Phi \stackrel{\text{def}}{=} \Sigma_{i\in I}\Phi_i = \{\langle i, A\rangle \mid i \in I, A \in \Phi_i\}$.*

A pair $\langle i, A\rangle \in \Phi$ is an assertion of judgement *i. We write $J_i(A)$ instead of $\langle i, A\rangle$.*

A judgement (single conclusioned) consequence relation *(JCR) $\rhd$ on $(\Phi_i)_i$ is a simple consequence relation on the set of formulæ Φ.*

Therefore, in this very general approach we add an extra degree of freedom in designing a consequence relation.

3.3.2 Multiple Consequence Relations

Another (and strictly related) approach is to consider "bidimensional consequence relations," that is sequents whose antecedents and consequents are *families* of (multi)sets of formulæ. These sequents can be arranged in a bidimensional fashion, as follows

$$\begin{array}{cc} \Gamma_1 & A_1 \\ \vdots & \vdots \\ \Gamma_n & \rhd \quad A_n \\ \vdots & \vdots \end{array}$$

or "linearized", separating each set by means of a semicolon:

$$\Gamma_1; \ldots; \Gamma_n; \ldots \rhd A_1; \ldots; A_n; \ldots$$

Each level/set corresponds to a different judgement, or a different consequence relation.

Let us formalize these notions. We begin with a definition of multiple Consequence Relation, which generalizes directly the notion of simple Consequence Relation.

Definition 3.10 (Multiple Consequence Relation) *Let I a set of indexes, and $(\Phi_i)_i$ a family of sets of formulæ, for $i \in I$, (possibly with repetitions). A* multiple (single conclusioned) consequence relation *(MCR) $\rhd$ on $(\Phi_i)_i$ is a family $(\rhd_i)_i$ of relations $\rhd_i \subseteq \left(\prod_{j\in I} \mathcal{P}_{\text{fe}}(\Phi_j)\right) \times \Phi_i$ such that each relation $\rhd_i$ satisfies the following:*

reflexivity: *$\forall \Gamma_j \in \mathcal{P}_{\text{fe}}(\Phi_j), j \neq i$, and $\forall A \in \Phi_i$: $(\Gamma_j)_{j<i}; A; (\Gamma_j)_{j>i} \rhd_i A$;*

(cross) transitivity: *$\forall \Gamma_j, \Gamma'_j \in \mathcal{P}_{\text{fe}}(\Phi_j), j \in I$, and $\forall A \in \Phi_k, k \neq i$, and $B \in \Phi_i$: if $(\Gamma_j)_j \rhd_k A$ and $(\Gamma'_j)_{j<k}; \Gamma'_k, A; (\Gamma'_j)_{j>k} \rhd_i B$ then $(\Gamma_j, \Gamma'_j)_j \rhd_i B$;*

monotonicity: $\forall \Gamma_j \in \mathcal{P}_{\mathrm{re}}(\Phi_j), j \in I$, *and* $\forall A \in \Phi_i, \forall B \in \Phi_k$,
if $(\Gamma_j)_j \rhd_i A$ *then* $(\Gamma_j)_{j<k}; \Gamma_k, B; (\Gamma_j)_{j>k} \rhd_i A$.

Although monotonicity and reflexivity are straightforward, transitivity deserves some explanations. It can be seen as a "cut" between possible different consequence relations; in fact, we cannot restrict ourselves to componentwise transitivity, because in deriving a certain consequence, we may make use of a lemma obtained on a different CR. Notice that each component of a MCR is a SCR on its own.

We do not investigate possible extensions of this definition, such as adopting multisets instead of sets of formulæ, or multiple-conclusion multiple consequence relations.

This approach is strictly related to the "judgements" approach. In fact, the following result holds:

Proposition 3.10 *Let I set of indexes, and Φ_i set for $i \in I$. Then, $\prod_{i\in I} \mathcal{P}_{\mathrm{re}}(\Phi_i)$ is isomorphic to $\mathcal{P}_{\mathrm{re}}(\Sigma_{i\in I}\Phi_i)$.*

Proof. It is easy to see that the maps

$$\begin{aligned} f: \quad & \textstyle\prod_{i\in I} \mathcal{P}_{\mathrm{re}}(\Phi_i) \to \mathcal{P}_{\mathrm{re}}(\Sigma_{i\in I}\Phi_i) \\ & (\Gamma_i)_{i\in I} \mapsto \{\langle i, A\rangle \mid i \in I, A \in \Gamma_i\} \end{aligned}$$

$$\begin{aligned} h: \quad & \mathcal{P}_{\mathrm{re}}(\Sigma_{i\in I}\Phi_i) \to \textstyle\prod_{i\in I} \mathcal{P}_{\mathrm{re}}(\Phi_i) \\ & \Delta \mapsto (\{A \mid \langle i, A\rangle \in \Delta\})_{i\in I} \end{aligned}$$

are computable bijections (since I is recusive). □

Due to this result, we can see a multiple consequence relation on a family $(\Phi_i)_i$ just as a simple consequence relation on a set of "tagged formulæ" $\{\langle i, A\rangle \mid i \in I, A \in \Phi_i\}$. Each index i can be seen, therefore, as a judgement, and $\langle i, A\rangle$ can be written $J_i(A)$ as well. Therefore, the multiple consequence relations and judgement consequence relation approaches are substantially equivalent, as shown by the following proposition:

Proposition 3.11 *Let I a set of indexes, and $(\Phi_i)_i$ a family of sets of formulæ, for $i \in I$. For each multiple consequence relation there is an equivalent judgement consequence relations, and vice versa.*

Proof. Let $(\rhd_i)_i$ be a MCR; then define the JCR $\rhd$ associated as $\Delta \rhd J_i(A) \iff h(\Delta) \rhd_i A$, where h is the function defined in Proposition 3.10. It is easy to see that $\rhd$ is a JCR, that is, it is reflexive, transitive and monotone on the set of assertions

On the other hand, if $\rhd$ is a JCR, define the MCR $(\rhd_i)_i$ associated as $(\Gamma_i)_i \rhd_i A \iff f((\Gamma_i)_i) \rhd J_i(A)$, where f is the function defined in Proposition 3.10. It is easy to see that $(\rhd_i)_i$ is a MCR, that is, it is reflexive and monotone componentwise, and enjoys cross transitivity. □

Schematicity is easily generalized to multiple consequence relations:

Definition 3.11 *Let $(\Phi_i)_{i\in I}$ a family of formulæ sets, and Sub_i a substitution schema on Φ_i for each $i \in I$. Let $(\rhd_i)_i$ on $(\Phi_i)_{i\in I}$ be a multiple consequence relation.*

We say that $(\rhd_i)_i$ is

- schematic relatively to Sub_j *if, when* $(\Gamma_i)_i \rhd_j A$ *then* $\forall (\sigma_i)_i$ *such that* $\sigma_i \in Sub_i$: $(\Gamma_i\sigma_i)_i \rhd_j A\sigma_j$.
- schematic relatively to $(Sub_i)_i$ *if it is schematic relatively to Sub_i for all $i \in I$.*

Truth-validity MCR for FOL

As an example, we examine a multiple consequence relation for First-Order Logic (for definitions, see Section 3.1.3.) Our MCR will be composed by two relations over the same set of formulæ, namely the language of first-order formulæ Φ. These relations are defined as follows:

$$\begin{aligned}\Gamma_1;\Gamma_2 \models_1 \varphi &\iff \forall\mathcal{M}.(\forall\rho.[\![\Gamma_1]\!]_{\mathcal{M}}\rho = T) \Rightarrow (\forall\rho.[\![\varphi]\!]_{\mathcal{M}}\rho = T)\\ \Gamma_1;\Gamma_2 \models_2 \varphi &\iff \forall\mathcal{M}.(\forall\rho.[\![\Gamma_1]\!]_{\mathcal{M}}\rho = T) \Rightarrow (\forall\rho.[\![\Gamma_2]\!]_{\mathcal{M}}\rho = T \Rightarrow [\![\varphi]\!]_{\mathcal{M}}\rho = T)\end{aligned}$$

The first relation, $\models_1$, acts as "validity" CR. From its point of view, assumptions have to be intended only as "theorems", valid for all environments; hence, it does not take care of "local assumptions." The second one, instead, discriminates between "theorems" and "local assumptions." We can indeed redefine the truth and validity consequence relations of Section 3.1.3 in terms of $\models_2$:

$$\begin{aligned}\Gamma \Vdash \varphi &\iff \Gamma;\Delta \models_1 \varphi\\ &\iff \Gamma;\emptyset \models_2 \varphi\\ \Gamma \models \varphi &\iff \emptyset;\Gamma \models_2 \varphi\end{aligned}$$

Monotonicity and reflexivity of each component is straightforward. Due to the cross-transitivity, there are four cases:

- "if $\Gamma_1;\Gamma_2 \models_1 \varphi$ and $\Gamma_1', \varphi; \Gamma_2' \models_1 \psi$, then $\Gamma_1, \Gamma_1'; \Gamma_2, \Gamma_2' \models_1 \psi$."
  This is simply the transitivity of $\Vdash$.
- "if $\Gamma_1;\Gamma_2 \models_2 \varphi$ and $\Gamma_1'; \Gamma_2', \varphi \models_2 \psi$, then $\Gamma_1, \Gamma_1'; \Gamma_2, \Gamma_2' \models_2 \psi$."
  This is simply the transitivity of $\models$.
- "if $\Gamma_1;\Gamma_2 \models_1 \varphi$ and $\Gamma_1', \varphi; \Gamma_2' \models_2 \psi$, then $\Gamma_1, \Gamma_1'; \Gamma_2, \Gamma_2' \models_2 \psi$."
  It is easy to see that this statement holds.
- "if $\Gamma_1;\Gamma_2 \models_2 \varphi$ and $\Gamma_1'; \Gamma_2', \varphi \models_1 \psi$, then $\Gamma_1, \Gamma_1'; \Gamma_2, \Gamma_2' \models_1 \psi$."
  This statement holds trivially, since $\models_1$ does not consider formulæ in Γ_2, Γ_2'.

3.4 Natural Deduction-style Proof Systems

In this section we briefly recall the Natural Deduction style of presentation of proof systems.

Before building an editor for a given logic, one of the paramount issues that the designer/implementor has to clarify is which style of presentation is best for actually "using" the logic, e.g. Hilbert, Natural Deduction (ND) or Gentzen (sequent) style?

From a practical point of view, Natural Deduction style systems are more suited to the practical usage — actually, they have been introduced by Gentzen because they "reflects as accurately as possible the actual logical reasoning involved in mathematical proofs" [Gen69]. These proof systems allow the user to reason "under assumptions" and go about

in developing a proof the way mathematicians normally reason: using hypotheses, formulating conjectures, storing and retrieving lemmata, often in top-down, goal-directed fashion. Moreover, Logical Frameworks based on Type Theory directly give arise to proof systems in Natural Deduction style. This follows directly from the "judgement-as-types" paradigm, and the fact that the typing systems of the underlying λ-calculi are in Natural Deduction style.

We write Natural Deduction rules and proofs in the *linearized* notation, hence "$\pi : \Gamma \vdash \varphi$" denotes a proof tree π whose premises and conclusion are Γ and φ respectively.

Following [Avr91], systems in Natural Deduction style can be defined as follows:

Definition 3.12 (Natural Deduction style) *A Natural Deduction style system in the language Φ is an axiomatic system for deriving sequents of the form $\Gamma \vdash \varphi$, for $\Gamma \subseteq \Phi, \varphi \in \Phi$, such that*

1. *For all $\varphi \in \Phi$, $\varphi \vdash \varphi$ is an axiom of the system;*
2. *For every rule of the system, the set of assumptions of the conclusion is a subset of the union of the assumptions of the premises. More formally, if a rule has $\Gamma_1 \vdash \varphi_1$, $\ldots$, $\Gamma_n \vdash \varphi_n$ as premises, and $\Gamma \vdash \varphi$ as conclusion, then $\Gamma \subseteq \bigcup_{i=1}^{n} \Gamma_i$.*

Usually, rules in ND-style calculi are defined by general *rule schemata* (*figures*, in Gentzen's terminology [Gen69]) of the form

$$\forall \Gamma_1, \ldots, \Gamma_n \frac{\Gamma_1, \Delta_1 \vdash \varphi_1 \ldots \Gamma_n, \Delta_n \vdash \varphi_n}{\Gamma_1, \ldots, \Gamma_n \vdash \varphi} C$$

which is usually written in the so-called "vertical form":

$$\frac{\begin{array}{ccc}(\Delta_1) & \ldots & (\Delta_n) \\ \vdots & & \vdots \\ \varphi_1 & \ldots & \varphi_n\end{array}}{\varphi}$$

where C is a possible *side condition*, that is a restriction on the applicability of the schemata. In the latter form, the "entailment" symbol disappears, and the universal quantification is made implicit; however, it should be clear that the intended meaning is always the former.

The Natural Deduction-style can be extended also to infinitary proof systems; it is sufficient allow for infinite rules. It should be clear, however, that a (Natural Deduction-style) proof, finite or infinite, is always a *well-founded proof of sequents*, labelled accordingly to the rules of the system. That is to say, every path leading out from the root (the conclusion) is finite (hence it ends with an assumption), although there may be an infinite number of paths. For instance, the Natural Deduction style ω-rule of ω-logic [Gir87b]

$$\frac{\varphi[0/x] \quad \varphi[1/x] \quad \ldots \quad \varphi[n/x] \quad \ldots}{\forall x \varphi}$$

has to be intended as

$$\frac{\Gamma \vdash \varphi[0/x] \quad \Gamma \vdash \varphi[1/x] \quad \ldots \quad \Gamma \vdash \varphi[n/x] \quad \ldots}{\Gamma \vdash \forall x \varphi}$$

In this way we can separate the finiteness of proofs from the finiteness of assumptions. We can define an infinitary system, complete with respect to the finitary truth consequence relation of first order logic $\models_{\mathcal{N}} \cap(\mathcal{P}_{\lesssim\omega}(\Phi) \times \varphi)$, by requiring that every sequent in (possibly infinitary) proofs are finite.

A rule with a side condition is said *impure*. Often, the "linearized" version is needed in order to state clearly the side conditions. Two classic examples are the following $\forall$-introduction rule for FOL [Gen69] and $\Box$-introduction rule for $S4$ [Pra65]:

$\forall$-I	$\frac{\varphi}{\forall x\varphi}$	x does not appear free in any assumption on which the derivation of φ depends.
$\Box$-I	$\frac{\varphi}{\Box\varphi}$	every formulæ on which φ depends, is boxed.

In this simple formulation, the side conditions are stated in an informal natural language. The linearize version allow us for spelling out precisely these conditions, as follows:

$$\forall\text{-I} \quad \frac{\Gamma \vdash \varphi}{\Gamma \vdash \forall x\varphi} \quad x \notin \mathrm{FV}(\Gamma)$$

$$\Box\text{-I} \quad \frac{\Gamma \vdash \varphi}{\Gamma \vdash \Box\varphi} \quad \forall\psi \in \Gamma \exists\psi'.\psi = \Box\psi'$$

However, not every side condition can be described rigorously by the linearized form. In fact, Avron identifies three degrees of impurity, listed here by increasing complexity:

1. Conditions on the *structure* of the sets of assumptions. Rules with this kind of conditions are also called *proof rules*. The best known example is the *necessitation rule*,
$$\Box'\text{-I}\frac{\Gamma \vdash \varphi}{\Gamma \vdash \Box\varphi}\Gamma = \emptyset$$

2. Conditions on the structure of the assumed formulæ. An example of this kind of impurity is the above-mentioned $\Box$-introduction rule of Prawitz' system $S4$, where every assumption is required to be boxed.

3. Conditions on the structure of the *proofs* themselves, that is, conditions on *how* the premises have been derived. An example of this kind of impurity is the $\Box$-introduction rule of Prawitz's "third version" system for $S4$:
$$\Box'\text{-I}\frac{\Gamma \vdash \varphi}{\Gamma \vdash \Box\varphi}(*)$$
where the requirement $(*)$ is "the derivation of φ from Γ contains a fringe of boxed formulæ", that is, "every path from φ to the assumptions contains a boxed formula".

Conditions of the third level are not *local*, in the sense that they cannot be decided only by looking at the sequents: we need to examine inside the whole proofs. Hence, a complete formalization of these side conditions needs of a formalism for representing the proofs themselves.

Another property usually enjoyed by ND-style calculi is an internal "symmetry": for each logical constructor, there is a (set of) introduction rules and a (induced) elimination rule; see [Pra65].

Part II

The Object Logics

Chapter 4

Structural and Natural Operational Semantics

In order to establish formally properties of programs, we have to represent formally their operational semantics. Several ways for presenting operational semantics of languages have been developed. An intuitive model of the way a program is executed is to think of a *machine state*, that is, a pair $\langle M, \sigma \rangle$ composed by a *control part* M and a data part σ. Computation is then represented by a sequence of these pairs. This approach is simple and intuitive; however, its low abstraction level does not allow for powerful proof techniques; we can reason only by induction on the length of computations. A detailed description of the various styles of presentation of operational semantics is beyond the scope of this thesis; see e.g. [Gun92, Win93, Plo81] for more details.

In this chapter, we aim to understand, from a logical point of view, how to take advantage of the Natural Deduction style for representing concisely operational semantics. This will lead us to the introduction of the *Natural Operational Semantics* (NOS) presentations. This style has both theoretical and practical advantages. Firstly, Natural Deduction style systems are interesting on their own. Moreover, Logical Frameworks naturally allow the user to reason "under assumptions" and go about in developing a proof in Natural Deduction fashion. Therefore, Natural Deduction presentations of logics are easier to encode faithfully in logical frameworks.[1]

Structure of the Chapter. This chapter is structured as follows. In Section 4.1 we recall and analyze the SOS paradigm from a logical point of view. In Section 4.1.1, we introduce the semantic counterpart of SOS proof systems in terms of consequence relations; in Section 4.1.2, we discuss the general features of the SOS paradigm. This discussion will yield the introduction of the *Natural Operational Semantics* (NOS) style of presentation (Section 4.2). In particular, the *bookkeeping* technique, which is at the base of NOS presentations, is introduced in Section 4.2.2.

A complete case study will follow: in Section 4.3 we examine a functional language which includes some imperative features (such as assignments) and whose semantics can been successfully described by using the NOS paradigm.

[1] Some of the results of this chapter have been previously presented in [Mic94].

A running Example. Along this chapter, we will often refer to a simple functional language $\mathcal{L}_{ALD}$, which is a language of **a**rithmetic expressions with **l**ocal **d**eclaration. It is composed by two syntactic classes: *Expr*, the class of expressions ranged over by M, m, and *Var*, the class of identifiers (ranged over by x, y), the former including the latter.

$$\begin{array}{lll} \mathit{Var} & x ::= & i_0 \mid i_1 \mid \ldots \\ \mathit{Expr} & M ::= & 0 \mid 1 \mid M + N \mid x \mid \textbf{let } x = M \textbf{ in } N \end{array}$$

Despite the extreme simplicity of this language, the comparision between the SOS and the NOS specifications for this language will be an elightening starting point for the understanding of the principles of the NOS style of presentation.

4.1 Structural Operational Semantics

A very successful style of presenting operational semantics is the one introduced by Gordon Plotkin and known as *Structural Operational Semantics* (SOS) [Plo81], and further improved by Gilles Kahn and his coworkers [Kah87, CDDK86, Des86]. The idea behind this approach is that all computational elaboration and evaluation processes can be construed as logical processes and hence can be reduced to the sole process of formal logical derivation within a formal system.

Example 4.1 The SOS for $\mathcal{L}_{ALD}$, as for any functional language, is a formal system for inferring assertions of the form $E(\rho, M, m)$, where m is the *value* of the *expression* M, and ρ is the *environment* in which the evaluation is performed—usually a finitary function mapping identifiers to values. The intended meaning of this proposition is "in the environment ρ, the evaluation of M gives m." Here we give some sample rules:

$$\frac{}{E(\rho, x, \rho(x))} \qquad \frac{E(\rho, N_1, n_1) \quad E(\rho, N_2, n_2)}{E(\rho, (N_1 + N_2), n)} n = n_1 + n_2 \qquad \frac{E(\rho, M, m) \quad E([x \mapsto n]\rho, N, n)}{E(\rho, (\textbf{let } x = M \textbf{ in } N), n)}$$

Usually, the assertion "$E(\rho, M, m)$" is written as "$\rho \vdash M \to m$", but the intended meaning of "$\vdash$" is *not* that of a Consequence Relation. In the SOS paradigm, the "turnstile" is used for keeping apart the environmental informations (i.e. what does not change during computation) from data structures which are modified by the computational process. □

4.1.1 Consequence Relations for SOS

From a logical point of view, a SOS specification is nothing but a proof systems for deriving assertions of the evaluation judgement, such as, say, "$E(\rho, M, m)$." However, in these derivations we usually keep a "theoremhood" approach: we do not take into account derivations from assumptions, but only derivations from the empty set. So, instead of "$\emptyset \vdash E(\rho, M, m)$", which means that "$E(\rho, M, m)$ can be inferred from the empty set", we write "$\rho \vdash M \to m$ holds." It is clear that the intended meaning of "$\vdash$" is not the same.

We may wonder, then, which is the semantic consequence relation represented by a SOS specification. This CR should be given in terms of a reference semantics, usually the denotational one, of the language. Such a denotational semantics would be a truth function $[\![\cdot]\!]$, which provides each evaluation assertion its truth value. For instance, in

the case of a **while** language, $[\![E(C,\sigma,\sigma')]\!] = \top$ if and only if $[\![C]\!]\sigma = \sigma'$, where $[\![C]\!]$ is a suitable denotational semantics of C (see, e.g., [Gun92, Win93, Sch86, Sch94]).

The immediate, naïve way for defining a CR is just to choose the least CR which allows for evaluation assertions also on the left-hand side of $\vdash$. This is exactly the free CR on the semantics of assertions (see Definition 3.2):

Proposition 4.1 *Let S be a SOS specification, and let Φ be the class of its evaluation assertions, ranged over by A. Let $[\![\cdot]\!] : \Phi \to \{\bot, \top\}$ a (denotational) semantics for Φ, sound and complete for S (i.e., $\vdash_S A \iff [\![A]\!] = \top$). Then, S represents the free CR on $[\![S]\!]$.*

Example 4.2 As we have seen in Example 4.1, the language $\mathcal{L}_{ALD}$ can be given an SOS specification, which is a system S for deriving evaluation assertions of the form "$E(\rho, M, m)$", where ρ ranges over *Env*, the set of finite functions from *Var* to close expressions (i.e. values) of *Expr*. This system features rules such as those of Example 4.1.

The evaluation judgement can be given a semantics in terms of the denotational semantics of the language. Let $[\![\cdot]\!]_D : \mathit{Expr} \to (\mathit{Var} \to D) \to D$ a denotational semantics for *Expr*, in a domain D. We define:

$$[\![E(\rho, M, m)]\!] \stackrel{\text{def}}{=} \begin{cases} \top & \text{if } [\![M]\!]_D\rho^* = [\![m]\!]_D\rho^* \\ \bot & \text{otherwise} \end{cases}$$

where $\rho^* : \mathit{Var} \to D$ is just $\rho^*(x) \stackrel{\text{def}}{=} [\![\rho(x)]\!]\rho^*$. The free CR $\models_D$ on $[\![\cdot]\!]_D$ is the following:

$$\begin{aligned} E(\rho_1, M_1, m_1), \ldots, E(\rho_n, M_n, m_n) \models_D E(\rho, M, m) \iff \\ ((\forall i = 1 \ldots n : [\![M_i]\!]\rho_i^* = [\![m_i]\!]\rho_i^*) \Rightarrow [\![M]\!]\rho^* = [\![m]\!]\rho^*) \end{aligned}$$

However, assumptions are not relevant, since usually only theorems are considered. Indeed, adequacy of $\vdash$ w.r.t. $\models_D$ restricted to theorems is just adequacy of operational semantics w.r.t. denotational one; in fact, $(\vdash) \cap (\{\emptyset\} \times \Phi) = (\models_D) \cap (\{\emptyset\} \times \Phi)$ holds iff $\forall \rho, M, m : \emptyset \vdash E(\rho, M, m) \iff [\![M]\!]\rho = [\![m]\!]\rho$. □

The assumptions of a free consequence relation, such as $\models_D$ can be fruitfully used in dealing with "incomplete" computations. A derivation of the evaluation judgement from assumptions can be seen as a "partial computation", where the assumptions represent the computations to be still performed, or performed by some external agent (i.e., linked libraries). In particular, the rules themselves can be given a direct meaning in terms of this semantic consequence relation, since a rule $\frac{\varphi_1, \ldots, \varphi_n}{\varphi}$ is sound with respect to a CR $\models$ iff $\varphi_1, \ldots, \varphi_n \models \varphi$.

For instance, the "+" rule above-mentioned (Example 1) is sound w.r.t. $\models_D$ since

$$\begin{aligned} \forall \rho, M_1, M_2, m_1, m_2 : \quad & [\![E(\rho, M_1, m_2)]\!]_D = \top \wedge [\![E(\rho, M_2, m_2))]\!]_D = \top \\ & \Rightarrow [\![E(\rho, (M_1 + M_2), \mathit{plus}(m_1, m_2))]\!]_D = \top \end{aligned}$$

Nevertheless, we will see next that assumptions in consequence relation can be used for carrying more information than the one in this case.

4.1.2 Analysis of Structural Operational Semantics

The SOS style of specification does not have many of the defects of other formalisms (such as automata and definitional interpreters), since it is syntax-directed, abstract and easy to understand. It has been proved to be very successful in various areas of theoretical computer science. It was studied in depth by Kahn, Despeyroux and many of their coworkers [Kah87, Des86, CDDK86], who improved further this formalism introducing the *Natural Semantics*. Even further improvements have been recently carried out in the *Extended Natural Semantics* by Hannan [Han93, MP91, HP92]. It has been used by Harper, Milner and Tofte with the name of *Relational Semantics* to give the operational semantics of ML [MTH90]. Operational semantics of most languages can be handled successfully in this way; see Plotkin's work for an extensive account [Plo81].

Of course, the more complex is the semantics of the language, the richer and more detailed are the data structures to be dealt with by the evaluation judgement. Evaluation of very simple languages, such as arithmetic expressions, do not need of any auxiliary structure. On the other hand, a complex language such as Pascal or C, featuring local variables, pointers, aliasing, recursive procedures and jumps, needs several auxiliary data structure in order to keep track of informations along the computations. In the SOS approach, these data structures are very close to those we adopt in defining the denotational semantics—if not the same:

- *environment functions*: functions from identifiers to denotable values; these can be values such as integers, reals, or locations, but also functions and procedures as well;
- *stores*: functions from locations to storable values;

By combining these data structures, we can give the SOS of several languages. The evaluation judgement has to be adapted accordingly, in order to take care of the extra informations. Here we list some of these judgements:

- $E(M, m)$, written "$M \to m$," for simple expression languages, with no variables;
- $E(\rho, M, m)$, written "$\rho \vdash M \to m$," for pure functional language;
- $E(C, \sigma, \sigma')$, written $\langle C, \sigma\rangle \to \sigma'$ for simple imperative languages with no procedures, pointers and other sources of aliasing; σ, σ' are *stores*;
- $E(\rho, C, \sigma, \sigma')$, written $\rho \vdash \langle C, \sigma\rangle \to \sigma'$ for imperative languages with aliasing;
- $E(\rho, \psi, C, \sigma, \sigma')$, written $\rho, \psi \vdash \langle C, \sigma\rangle \to \sigma'$ for imperative languages with procedure declarations; ψ is the *procedural environment*;
- $E(\rho, M, \sigma, m, \sigma')$, written $\rho \vdash \langle M, \sigma\rangle \to \langle m, \sigma'\rangle$, for functional languages with imperative features, such as ML, or imperative languages like C, whose commands return a value.

There are two main drawbacks in the SOS specification paradigm. The first one is about the explicit presence of environmental structures and stores in evaluation assertions.

- the abstraction power is limited: a function which maps identifiers to values amounts to von Neumann's computer's memory.

- Modularity is limited. Modularity of semantic descriptions is an ongoing area of research—see e.g. [Mog93, WF91, Cen94]. As we have said before, in considering extensions of the language we may be forced to change the evaluation judgement itself. For instance, by adding to the simple language $\mathcal{L}_{ALD}$ some imperative features (such as references), we ought to change the judgement into the form $\rho \vdash \langle M, \sigma \rangle \rightarrow \langle m, \sigma' \rangle$. Hence, previous rules and derivations are not any more compatible with the new assertion. Rules have to be rephrased, and derivations have to be carried out again, even if they do not deal with any of the added structures. A simple and common extension which causes such problems is the introduction of new kinds of identifiers and denotable objects. Consider, for example, the introduction of procedure identifiers and procedures, in the simple **while** language. In this case, the evaluation judgement becomes $\psi \vdash \langle C, \sigma \rangle \rightarrow \sigma'$, in place of $\langle C, \sigma \rangle \rightarrow \sigma'$, for which rules previously introduced are useless. In general, any new class of denotable objects yields the introduction of a new environmental structure, and hence previous rules are no more appliable.

- The system lacks conciseness: environments appear in all rules but are seldom used. For instance, in the "+" rule, ρ plays no rôle: it is merely transferred from conclusion to premises (in a top-down proof development). The environment is effectively used only when we are dealing with identifiers, that is when we either declare an identifier or evaluate it, e.g. in the **let** and the variable-evaluation rules.

- In order to reason formally about properties of operational semantics, it is necessary to encode the formal system into some proof-editor/checker. However, in most proof assistants, representation of functions (such as the environments) can be rather cumbersome, and mechanized reasoning about these encodings can be very hard.

The second remark is that since we usually do not care of "partial evaluations," the free consequence relation which arises from denotational semantics is not useful—the assumptions carry no information during evaluations. In other words, as far as theorems are concerned, the essence of consequence is not exploited: in fact, assumptions play no rôle.

Therefore, one may wonder whether it is possible to take full advantage of the extra degree of expressiveness given by the assumptions. Can it be used in overcoming some of the drawbacks of environmental structures?

A possible answer to this question is the *Natural Operational Semantics*, which is presented in the next section.

4.2 Natural Operational Semantics

The *Natural Operational Semantics* formalism (NOS) has been introduced by Burstall and Honsell in [BH90], as a refinement of the Natural Semantics originally proposed by Kahn and his coworkers [Des86, Kah87]. This formalism arises if we take seriously the possibility of deriving under assumptions assertions in Natural Semantics, i.e. using hypothetico-general judgements in the sense of Martin-Löf [NPS92, NPS90]. It is based in fact on Gentzen's Natural Deduction style of proof [Gen69, Pra65]: hypothetical premises are used to make assumptions about the values of variables. Thus, instead of evaluating an expression within a given environment, we compute its value under a set of assumptions

on the values of its free variables. In other words, we replace explicit environments with implicit contextual structures, that is, the hypothetical premises in Natural Deduction.

We proceed as follows. In section 4.2.1 we present the NOS for $\mathcal{L}_{ALD}$. In section 4.2.2 we will generalize this approach introducing the *bookkeeping technique*. In section 4.2.3 we will analyse the main features of the NOS paradigm; finally, in section 4.2.4 we will give a semantical interpretation of NOS specifications by means of multiple consequence relations.

4.2.1 An example: Natural Operational Semantics for $\mathcal{L}_{ALD}$

Let us consider the functional language with two syntactic classes, *Expr* and *Var* of Example 4.1. We have seen that the SOS of this language is a system for deriving ternary judgements of the form $E(\rho, M, m)$, written simply $\rho \vdash M \rightarrow m$. Instead, in the NOS paradigm the judgements can be simplified to those of the form $M \Rightarrow m$, whose reading is "the value of expression M is m". There are no more contextual structures: the predicate is $\Rightarrow \subset \mathit{Expr} \times \mathit{Expr}$.

These assertions can be inferred using a Natural Deduction style proof system. The evaluation of the expression M to the value m can be represented by the following derivation in N.D. style:

$$\begin{array}{c} \Gamma = \{x_1 \Rightarrow n_1, \ldots, x_k \Rightarrow n_k\} \\ \triangledown \\ \pi \\ \triangledown \\ M \Rightarrow m \end{array} \qquad \text{written} \quad \pi : \Gamma \vdash M \Rightarrow m$$

where the hypotheses $\Gamma = \{x_1 \Rightarrow n_1, \ldots, x_k \Rightarrow n_k\}$ $(k \geq 0)$ can be interpreted as a set of variable bindings: the value of the variables involved in the evaluation of M. This derivation can be read as "in every environment which satisfies the assumptions in Γ, M is evaluated to m." This means that, given an environment ρ s.t. $\forall (x \Rightarrow m) \in \Gamma : \rho(x) = m$, there is a derivation of $\rho \vdash M \rightarrow m$ in the corresponding SOS proof system. An assumption about the value of a variable can be discharged when it is valid locally to a subcomputation. For instance, in the case of local declarations, in order to evaluate **let** $x = N$ **in** M, we can evaluate M assuming that the value of x is the same as that of N. This extra assumption is not necessary for evaluating **let** $x = N$ **in** M, so it can be discharged. Therefore, the rules of Example 4.1 are changed as follows:

$$\frac{\begin{array}{c}(x' \Rightarrow n) \\ \vdots \\ N \Rightarrow n \quad M' \Rightarrow m'\end{array}}{\textbf{let } x = N \textbf{ in } M \Rightarrow m}\, \mathsf{EC}(x, M, m) \qquad \frac{N_1 \Rightarrow n_1 \quad N_2 \Rightarrow n_2}{N_1 + N_2 \Rightarrow n} n = n_1 + n_2$$

where $\mathsf{EC}(x, M, m)$ is a typographic abbreviation for the *Eigenvariable Condition*:

$$\mathsf{EC}(x, M, m) \equiv \begin{array}{l}\text{"}M', m' \text{ are obtained from } M, m \text{ respectively by replacing } \textit{all} \text{ the} \\ \text{occurrences of } x \text{ with } x', \text{ which appears neither in } x,\ M,\ m \text{ nor in} \\ \text{any live assumption different from } (x' \Rightarrow n)\text{"}.\end{array}$$

where by "live assumptions" we mean the assumptions which are not discharged from the proof context.[2]

Actually, in the simple case of $\mathcal{L}_{ALD}$, there is no need of replacing variables in m, since it is always a numeral. This will not hold in richer languages, which allow for more complex values such as *closures* (see Section 4.3.)

This truly N.D. approach has the benefit that all the rules which do not refer directly to identifiers appear in a simpler form than those in SOS style: no environment appears (see the "+" rule). Moreover there is no specific rule for the evaluation of identifiers.

The **let** rule is the rule in which the whole power of the ND style appears whose reading is "if n is the value of N and, assuming the value of x is n then m is the value of M, then the value of **let** $x = N$ **in** M is m." The renaming of the local variable is necessary since in the following naïve rule for **let**

$$\frac{\begin{array}{cc} & (x \Rightarrow n) \\ & \vdots \\ N \Rightarrow n & M \Rightarrow m \end{array}}{\textbf{let } x = N \textbf{ in } M \Rightarrow m}$$

the local assumption can clash with a previous assumption on x which is valid globally, yielding incorrect derivations such as the following:

$$\frac{\dfrac{}{0 \Rightarrow 0} \qquad \dfrac{\dfrac{}{1 \Rightarrow 1} \qquad (x \Rightarrow 0)_{(1)}}{\textbf{let } x = 1 \textbf{ in } x \Rightarrow 0}}{\textbf{let } x = 0 \textbf{ in let } x = 1 \textbf{ in } x \Rightarrow 0}(1)$$

The difficulty of avoiding the capturing of local variables can be overcome by making explicit the textual substitution of variables in local evaluations [BH90]. This is similar to Gentzen's notion of *Eigenvariable* [Gen69]. Recall the ∃-ELIM rule:

$$\frac{\exists x.A \qquad \begin{array}{c}(A') \\ \vdots \\ B\end{array}}{B}$$

A' is obtained from A by replacing all the occurrences of x with x', where x' does not occur neither in any of A, x, B, nor in any assumption different from A'.

Similarly, in evaluating **let** $x = N$ **in** M, we have to replace all the occurrences of x in M with a new identifier never used before, say x', which will be bound to the value of n.

Why not using higher-order abstract syntax?

Another attempt to overcome the capturing of local variables could be that of using *higher-order abstract syntax* à la Church (see Chapter 2). For example, the construct **let** $x = N$ **in** M could be "compiled" to **let** N $(\lambda x.M)$, where **let** : $Expr \to (Expr \to$

[2] This terminology is due to J.-Y. Girard and his co-workers.

Expr) $\to$ *Expr*. In this case, the evaluation rule can be stated as follows:

$$\frac{\begin{array}{c}(x \Rightarrow n)\\ \vdots \\ N \Rightarrow n \quad M \Rightarrow m\end{array}}{\textbf{let } x = N \textbf{ in } M \Rightarrow m} \; x \text{ does not appear free in any live hypothesis}$$

It should be noticed, however, that the applicability of this rule relies upon an implicit α-conversion of bound variables: we need to assume that

$$\textbf{let } x = N \textbf{ in } M \quad \equiv \quad \textbf{let } y = N \textbf{ in } M[y/x]$$

for $y \notin \mathrm{FV}(M) \setminus \{x\}$; otherwise, without this assumption we cannot deal with nested declarations of the same variable, such as in the evaluation of **let** $x = 0$ **in let** $x = 1$ **in** x. The α-conversion of bound variables is obtained "for free" if the **let** expressions are represented by using higher-order abstract syntax, because

$$(\textbf{let } x = N \textbf{ in } M) = (\textbf{let } N\ \lambda x.M) \equiv (\textbf{let } N\ \lambda y.M[y/x]) = (\textbf{let } y = N \textbf{ in } M[y/x])$$

As we will see in Chapters 10, 11, this technique has been proved to work extremely well for purely functional languages. In fact, the resulting paradigm, called *Extended Natural Semantics* (ENS) [Han93], is commonly adopted in representing the semantics of functional languages (see [AHMP92] for a treatment of this in the context of λ-calculus and [Han88, MP91, HP92] of more general functional languages). However, the HOAS approach cannot be applied directly in the case of languages with imperative features. In fact, it easily yields semantic inconsistencies, since it treats identifiers as placeholders for expressions. This is correct in pure functional languages, but does not hold in imperative languages, as we are going to see (cf. also the discusssion about Dynamic Logic, in Section 3.2.3).

Suppose there is a sort *Comm* of commands (such as the minimal **while** language), and a new expression constructor $[\cdot]\cdot : \mathit{Comm} \times \mathit{Expr} \to \mathit{Expr}$, which applies commands to expressions (see Section 4.3.1, and [BH90]); $[C]M$ intuitively means "execute C and then evaluate M". An assignment command $x := M$ can be seen as an *explicit substitution*; hence, one is tempted to adopt HOAS and to translate an expression like $[x := N; C]M$ into **let** $N\ \lambda x.[C]M$. However, this approach is doomed to fail because there is no direct representation of loops: in order to translate an expression of the form

$$[\textbf{while } b \textbf{ do } x := N; C]M,$$

we should unfold *statically* the **while** as many times as we are going to repeat the loop, which is impossible. Hence, in presence of commands we need to keep assignments (explict substitutions) in expressions, delaying their evaluation at the run-time.

The use of higher-order abstract syntax raises other problems, beside the one presented here. We just recall the lacking of induction principles (see [DFH95] and Chapter 11); we refer also to [AHMP92] for more difficulties in handling Hoare logic.

4.2.2 The Bookkeeping Technique

In this section, we introduce the *bookkeeping* technique, as a generalization of the previous example.

The treatment of local variables we have seen in the case of a simple functional language in the previous section can be actually generalized to *any* kind of local binding. Let us consider again the enlightening **let** rule:

$$\frac{\begin{array}{c}(x' \Rightarrow n)\\ \vdots \\ N \Rightarrow n \quad M' \Rightarrow m'\end{array}}{\textbf{let } x = N \textbf{ in } M \Rightarrow m}\ \mathsf{EC}(x, M, m)$$

where $\mathsf{EC}(x, M, m)$ is as before. This treatment of local variables correctly obeys the standard stack discipline of languages with static scoping: when we have to define a local variable, we allocate a new cell (represented by x', a new variable) where we store the local value (this is achieved by assuming $x' \Rightarrow n$). This allocation is active only during the evaluation of M (the derivation tree of $M' \Rightarrow m'$); then, the cell is disposed of (x' does not occur in any other place).

Any static-scope declaration can be represented by means of a specific rule, following the pattern of the **let** rule above. That is, informations we usually put in the left-hand side of SOS judgements, such as the ρ, ψ of "$\rho \vdash M \to m$" and "$\rho, \psi \vdash \langle C, \sigma\rangle \to \sigma'$", can always be bookkept in the proof context of a Natural Deduction proof in NOS.

In some cases, this bookkeeping can be implemented by means of the same evaluation judgement, as in the case of the simple functional language we have seen above. More generally, however, it is easier to adopt an *ad hoc* judgement for each environmental information we need to keep during evaluation.

Definition 4.1 (Bookkeeping judgement) *Let $\mathcal{L}$ be a language, and let Φ a sort of $\mathcal{L}$, and* Var *the sort of identifiers for Φ. A* bookkeeping judgement *for Φ is a judgement $\mapsto$: Var $\times \Phi$. An assertion $x \mapsto A$ is called a* binding.

The only purpose a bookkeeping judgement is used for is to keep the link between a variable and the associated value in the proof context. There is indeed a strict relation between environment maps of SOS and sets of bindings:

Definition 4.2 (Valid Contexts and Representation) *Let $\mathcal{L}$ be a language, and I an index family; for $i \in I$, let*

- *Φ_i be a syntactic sort of $\mathcal{L}$ and Var_i the sort of variables for Φ_i;*
- *$\rho_i : \mathrm{Var}_i \to \Phi_i$ be an environment (finite map) for Φ_i;*
- *$\mapsto_i$: $\mathrm{Var}_i \times \Phi_i$ be a bookkeeping judgement for Φ_i.*

Let Γ be a set of bindings for $(\Phi_i)_i$, that is a set of assertions of the form "$x \mapsto_i m$". We say that Γ is a valid context *if $\forall (x \mapsto_i m), (x' \mapsto_i m') \in \Gamma$ if $x = x'$ then $m = m'$.*

A valid context Γ represents the environments $(\rho_i)_i$ *if $\forall i \in I \forall x \in \mathrm{dom}(\rho).\exists (x \mapsto_i m) \in \Gamma.\rho_i(x) = m$.*

We say that the environments $(\rho_i)_i$ satisfy *a valid context Γ (written $(\rho_i)_i \vdash \Gamma$), if $\forall (x \mapsto_i m) \in \Gamma.\rho_i(x) = m$.*

Notice that there are *no* introduction rules for bookkeeping judgements: bindings are

- assumed in (or, in a bottom-up approach, discharged from) the context of a subderivation when we need to evaluate a local definition;
- premises of rules, where identifiers have to be evaluated.

Care has to be taken in hypothesis-discharging rules, in order to preserve validity of contexts. We have to ensure that at every time, no more than one value is associated to every variable. This can be obtained by adapting the *eigenvariable condition*: the local binding is assumed on a new variable (locally introduced), and renaming accordingly the variables occurring in the evaluation judgement.

In fact, following these guidelines any SOS system can be turned into a NOS specification, as it is stated by the following *bookkeeping principle*:

Bookkeeping Principle

Any SOS system S induces an equivalent NOS system S' which is obtained by

- dropping environmental structures from evaluation judgements;
- introducing a bookkeeping judgement for each environmental structure;
- in every rule,
  - premises whose environments are obtained by extending the environment of conclusion, are turned into binding-discharging subderivations;
  - evaluation on an environment structure are replaced by the corresponding binding.

The resulting system is equivalent, in the sense that there is a bijection between proofs π of SOS and derivations in NOS whose proof context represents the environment structures of the derived judgement of π.

Remark 1. So far, we have not faced explicitly the case of higher-order languages, which allow for functions as result values. In such cases, particular care has to be taken in evaluating λ-abstractions: the corresponding results should be *closures*, i.e. an expression representing the function together with the contextual information for their evaluation. As we will see in Section 4.3, NOS can easily and efficently deal with closures, by using an auxiliary judgement.

Remark 2. The way a SOS specification is turned into the equivalent NOS one resembles very closely a well-known transformation of Proof Theory: the construction of a Natural Deduction style system from a single-conclusion sequent style system. In fact, the logical analogy between these two syles of semantics and logical paradigms can be stated as

$$\frac{\textbf{Single-Conclusion Sequent Calculus}}{\textbf{Natural Deduction}} = \frac{\textbf{Structural Operational Semantics}}{\textbf{Natural Operational Semantics}}$$

SOS	NOS
$\dfrac{}{\rho \vdash x \to \rho(x)}$	$\dfrac{x \mapsto m}{x \Rightarrow m}$
$\dfrac{\rho \vdash N_1 \to n_1) \quad \rho \vdash N_2 \to n_2}{\rho \vdash (N_1 + N_2) \to n} n = n_1 + n_2$	$\dfrac{N_1 \Rightarrow n_1 \quad N_2 \Rightarrow n_2}{N_1 + N_2 \Rightarrow n} n = n_1 + n_2$
$\dfrac{\rho \vdash M \to m \quad [x \mapsto n]\rho \vdash N \to n}{\rho \vdash (\textbf{let } x = M \textbf{ in } N) \to n}$	$\dfrac{\begin{array}{c}(x' \mapsto n) \\ \vdots \\ N \Rightarrow n \quad M' \Rightarrow m'\end{array}}{\textbf{let } x = N \textbf{ in } M \Rightarrow m}\ (*)$

$(*)$ = "M', m' are obtained from M, m respectively by replacing *all* the occurrences of x by x', which does not appears neither in x, M, m nor in any live assumption different from $(x' \mapsto n)$".

Figure 4.1: SOS and NOS for $\mathcal{L}_{ALD}$.

Example 4.3 Following this technique, we can turn the SOS semantics of $\mathcal{L}_{ALD}$ into the equivalent NOS one by introducing the bookkeeping judgement $\mapsto$: $Var \times Expr$. Both the SOS and the induce NOS specification are presented in Figure 4.1.

It is easy to see that these specifications are strictly equivalent:

Proposition 4.2 *Let $\rho : Var \to Expr$ be an environment and Γ a context representing ρ. Then, $\rho \vdash_{SOS} M \to m$ iff $\Gamma \vdash_{NOS} M \Rightarrow m$. Moreover, this correspondence is bijective, that is, there exists a bijective function*

$$f \quad : \quad \{\pi \mid \pi : \rho \vdash_{SOS} M \to m\} \to \{\pi' \mid \pi' : \Gamma \vdash_{NOS} M \Rightarrow m\}$$

from proofs in SOS to derivations in NOS.

□

4.2.3 Analysis of Natural Operational Semantics

It is easy to see that the NOS paradigm overcomes many of the drawbacks of Structural Operational Semantics, still retaining all the advantages.

Advantages

Indeed, a NOS specification is syntax-directed, abstract and easy to understand at least as much as the corresponding SOS specification. Moreover, the NOS paradigm improves SOS since environment structures are cancelled from evaluation judgements by taking full advantage of assumptions and of the assumption-discharging mechanism of Natural Deduction. This has mainly two consequences.

Firstly, rules are more concise, because rules do not deal any more with environmental data, unless they are either declaration or evaluation rules. This frees the user of the burden of handling with growing structures all along the derivation.

Secondly, modularity is improved: if we add a new declaration construct, that is a new kind of binding, we need simply to add the bookkeeping judgement and the evaluation rule concerning the new construct. Previous rules and derivations are still valid in the extended system.

Another important advantage of NOS specifications is that, as any Natural Deduction-style system, they are easier to encode in Logical Frameworks based on Type Theory. We will face this problem in Chapter 12.

Bookkeeping and Tennent's Principles of Abstraction and Qualification

There is a strict relation between the bookkeeping technique, which is at the base of Natural Operational Semantics, and two of the main principles of programming languages design, due to Tennent [Ten81, Sch94]:

Abstraction: Phrases of any semantically meaningful syntactic class may be named.

Qualification: Any semantically meaningful syntactic class may admit local definitions.

These principles are strictly related to our techniques. The first one states substantially that every meanigful class can be denoted by a name; the second states that every phrase can be subject to a local declaration, e.g. can occur inside of the scope of a **let**.

Schmidt has reduced the Abstraction principle to the more general principle of

Record Introduction: Phrases of any semantically meaningful syntactic class may be components of records.

where by *record* we mean a set of bindings [Sch94, p.66]. Indeed, a valid environment is nothing but a record, and the bookkeeping technique correctly implements the introduction of new bindings. In other words,

> the technique of bookkeeping is the implementation (in Natural Deduction form) of Tennent's Abstraction and Qualification Principles.

Therefore, the operational semantics of a program constructor designed accordingly to these principles can be correctly represented in NOS by means of a bookkeeping judgement and relative binding-discharging and evaluating rules.

Limitations

Not every auxiliary data structure of SOS can be represented by means of assumptions in the NOS, as well as not every Sequent Calculus system can be translated neatly in Natural Deduction style. The structural rules implicit in Natural Deduction-style systems of monotonicity of hypothesis imply that only informations which can be dealt with using a static scoping discipline can be reduced to assumptions. In particular, a side-effect assignment of pointers which induces variables aliasing (or sharing) is difficult to encode. In fact, we cannot retrace from the context all the bindings which are involved on a set of shared variables whenever one of them changes its value.

In fact, informations which change during the evaluation process is usually kept in "stores". Stores are closer to a type of computation result than to an environment in

which computations take place. In the SOS paradigm, this is exhibit by putting stores on the right-hand side of "$\vdash$", like in the case $\rho \vdash \langle C, \sigma\rangle \to \sigma'$. Similarly, in the NOS paradigm, stores have to be kept in the evaluation judgement. For instance, in the case of imperative languages the evaluation judgement would be $\Rightarrow$: $\mathit{Stores} \times \mathit{Comm} \times \mathit{Stores}$. Of course, this difficulty rises again the modularity problems we have overcome in the case of environment; for instance, if we extend a pure functional language by adding some sharing constructor (such as locations and references), previous NOS specifications and derivations are no longer valid.

In languages which do not allow sharing, however, assignments can be reduced to definitions of new variables [Don77]. Therefore, we focus on this kind of languages. Namely, those whose semantics can be defined without using both environment and store. These comprise all purely functional languages, but also some interesting extensions of these which have genuinely imperative features. This is in fact our thesis: only languages whose denotational semantics is definable by using only the notion of environment can be conveniently handled using NOS. In the following we describe some of these languages.

4.2.4 Consequence Relations for NOS

In this section we give a semantical interpretation of NOS systems.

As we have described before, in a NOS derivation of $\Gamma \vdash \varphi$ the formulæ in Γ are usually of a different sort from the one of φ. Indeed, φ is an evaluation judgement while Γ is composed only of bindings, which can be seen as constraints on the environmental structures we will use in the evaluation. In this setting, "$\vdash$" is not a consequence relation in the sense of Definition 3.1, since it lacks both reflexivity and transitivity.

Anyhow, we can define an appropriate Multiple Consequence Relation for NOS. We split the unique context Γ in several subcontexts, one for each bookkeeping judgement. So, Γ_i will be composed only by the bindings of the i-th bookkeeping judgement. Each of these contexts will represent a different environment, or better, each environment will satisfy a different context, as follows:

Definition 4.3 (MCR for NOS) *Let $\mathcal{S}$ be a NOS systems dealing with n bookkeeping judgements $\mapsto_i$, and m evaluation judgements. Let each evaluation judgement be given a truth semantics which takes n environments, as follows:*

$$[\![\cdot]\!]_j \quad : \quad \mathit{Env}_1 \to \ldots \to \mathit{Env}_n \to \{\bot, \top\} \qquad\qquad j = 1, \ldots, m$$

We then define the MCR $(\models_k)_{k\in\{1,\ldots,m+n\}}$ for $\mathcal{S}$ as follows:
for $i = 1 \ldots n$:

$$\Gamma_1; \ldots; \Gamma_n; \Delta_1; \ldots; \Delta_m \models_i (x \mapsto_i n) \quad \Longleftrightarrow \quad (x \mapsto_i n) \in \Gamma_i$$

for $j = 1 \ldots m$:

$$\begin{array}{lll} \Gamma_1; \ldots; \Gamma_n; \Delta_1; \ldots; \Delta_m \models_{n+j} \varphi & \Longleftrightarrow & \forall(\rho_i)_i. \\ & & (\forall i = 1, \ldots, n : \rho_i \models \Gamma_i) \Rightarrow \\ & & (\forall j = 1, \ldots, m : [\![\Delta_j]\!]_j \rho_1 \ldots \rho_n = \top) \Rightarrow \\ & & [\![\varphi]\!]_j \rho_1 \ldots \rho_n = \top \end{array}$$

It is easy to prove that this family of relations form a MCR. It is worthwhile noticing that, beside the n contexts for the n bookkeeping judgements, there are m new contexts, namely the Δ's. The j-th context Δ_j contains the assumptions the j-th evaluation judgement. From the point of view of operational semantics, these assumptions correspond to *external* or *system calls*: the only thing we care of these computations is their result, no matter how they are performed.

In the case of $\mathcal{L}_{ALD}$, by applying the above definition, we obtain

$$\begin{aligned} \Gamma;\Delta \models_1 x \mapsto t \quad &\iff \quad (x \mapsto t) \in \Gamma \\ \Gamma;\Delta \models_2 (M \Rightarrow m) \quad &\iff \quad \forall\rho.(\rho \models \Gamma) \Rightarrow \\ &\qquad (\forall(N \Rightarrow n) \in \Delta.[\![N]\!]\rho = [\![n]\!]\rho) \Rightarrow [\![M]\!]\rho = [\![m]\!]\rho \end{aligned}$$

which is usually applied with $\Delta = \emptyset$; in this case, we recover the intended semantics of $\vdash$:

$$\Gamma;\emptyset \vdash M \Rightarrow m \quad \iff \quad (\forall(x \mapsto t) \in \Gamma : \rho(x) = t) \Rightarrow [\![M]\!]\rho = [\![m]\!]\rho$$

4.3 A case study: the language $\mathcal{L}_P$

In this section, we examine a functional language extended with imperative features as assignments which give it an imperative flavor. Its semantics can been successfully described by using the NOS paradigm. We give its syntax, its NOS and denotational semantics (see Appendix A) and we prove that the former is adequate w.r.t. the latter. Finally, we will discuss the relation between the NOS and a SOS description.

There are mainly two reasons for this study. Firstly, this study aim also to investigate the features of a language whose semantics can be successfully described by means of the NOS paradigm, without introducing cumbersome datastructures for representing stores.

Secondly, we will see that the NOS paradigm is in general feasible, but in practice, developing a NOS specification for a nontrivial language may be quite cumbersome. In fact, the language may be affected by peculiar idiosyncrasies, which yield slight modifications of the standard bookkeeping rules introduced in Section 4.2.2. For instance, a common variation is the introduction of extra judgements and extra datastructures, beside those of the language, in order to record the informations in the environment. This process will be explained in detail.

4.3.1 Syntax and Semantics

$\mathcal{L}_P$ is an untyped λ-calculus extended by a set of structured commands (Figure 4.2). These commands are embedded into expressions using the "modal" operator $[\mathbf{on}\ \cdot\ \mathbf{do}\ \cdot]\cdot$. The expression

$$[\mathbf{on}\ x_1 = M_1; \ldots; x_k = M_k\ \mathbf{do}\ C]M$$

can be read as:

> execute C in the environment formed only by the bindings $x_1 = M_1; \ldots; x_k = M_k$; use resulting values of these identifiers to extend the global environment in which M has to be evaluated, obtaining the value of the entire expression.

$$
\begin{array}{rrl}
\textit{Id} & x ::= & i_0 \mid i_1 \mid i_2 \mid i_3 \mid \ldots \\
\textit{Expr} & M ::= & 0 \mid \textit{succ} \mid \textit{plus} \mid \textit{true} \mid \textit{false} \mid \leq \mid \textit{nil} \mid M :: N \mid \textit{hd} \mid \textit{tl} \\
 & & \mid x \mid \textbf{lambda}\ x.M \mid MN \mid \textbf{let}\ x = M\ \textbf{in}\ N \\
 & & \mid \textbf{letrec}\ f(x) = M\ \textbf{in}\ N \mid [\textbf{on}\ R\ \textbf{do}\ C]M \\
\textit{ProcId} & P ::= & p_1 \mid p_2 \mid p_3 \mid \ldots \\
\textit{Decl} & R ::= & \langle\rangle \mid x = M; R \\
\textit{Commands} & C ::= & \textbf{nop} \mid x := M \mid C; D \\
 & & \mid \textbf{if}\ M\ \textbf{then}\ C\ \textbf{else}\ D \mid \textbf{while}\ M\ \textbf{do}\ C \\
 & & \mid \textbf{begin new}\ x = M;\ C\ \textbf{end} \\
 & & \mid \textbf{procedure}\ P(x, y) = C\ \textbf{in}\ D \mid P(x, M)
\end{array}
$$

Figure 4.2: $\mathcal{L}_P$, a functional language with commands and procedures.

The command C cannot have access to "external" variables other than $x_1 \ldots x_k$, so all possible side effects are concerned with only these variables. Moreover, the entire **on-do** expression above does not have any side effect: all environment changes due to C's execution are local to M.

$\mathcal{L}_P$ allows us to declare and use procedures. For the sake of simplicity, but without loss of generality, these procedures will take exactly two arguments. The first argument is passed by value/result, the second by value [Plo81]. Furthermore, the body of a procedure cannot access global variables, but only its formal parameters (and locally defined identifiers, of course). This means that when $P(x, M)$ is executed within the scope of the declaration **procedure** $P(y, z) = C$ **in** D, C is executed in the environment formed by only two bindings: $\{y \Rightarrow n, z \Rightarrow m\}$, where n, m are the values of x, M respectively. After C's execution, the new value of y is copied back into x. So, $P(x, M)$ can affect only x.

The restriction on global access forbids sharing of identifiers, so there is no need for a store. This does not drastically reduce the expressiveness of the imperative language. Donahue has shown that in this case, the call-by-value/result is a "good" simulation of the usual call-by-reference [Don77].

In [BH90] a different definition of procedure is given. There, procedures parameters are passed only by value, but procedures can have access to global variables. However, there is a problem with this approach, since the Natural Deduction-style treatment of procedures does not immediately lend itself to support side effects on global variables. In fact, procedure executions may have not the intuitive meaning, in that approach; for instance, the expression

$$[\textbf{on}\ x = 0\ \textbf{do procedure}\ P(z) = (x := z)\ \textbf{in}\ x := 1; P(\textit{nil})]x$$

would be evaluated to 1 instead of *nil*, which is clearly counter-intuitive. This is due to the fact that the assignment made by $P(\textit{nil})$ on the global variable x is local to the environment of the procedure itself. In fact, executions of such procedures leave the global environment unchanged.

In Section A.2.1 we give the denotational semantics for the language $\mathcal{L}_P$. Domains are introduced to represent all the entities we have defined. This semantics is self-explanatory. We follow the usual syntax [Gun92, Sch86, Win93]; $\underline{\lambda}$ denotes the *strict abstraction*: for each meta-expression M with free variable x on pointed domain D, $(\underline{\lambda}x.M)\bot = \bot$. Furthermore, $\underline{\underline{\lambda}}$ is the *double-strict abstraction*: for each meta-expression $M \neq \bot$ with free variable x on pointed domain D with both $\bot$ and $\top$, $(\underline{\underline{\lambda}}x.M)\bot = \bot, (\underline{\underline{\lambda}}x.M)\top = \top$. The $\top$ element represents the "run-time error" condition (i.e. an unimplemented trap, a division by 0, an array index out of range, etc.).

We use the standard domains. The domains used are *Unit* (the set composed by only one point), *Truth* (the boolean set composed by two points, *true* and *false*), $\mathcal{N}$ (the set of natural numbers).

4.3.2 Natural Operational Semantics

The complete NOS formal system for $\mathcal{L}_P$ consists of 67 rules; see Appendix A.1.1.

In order to deal with λ-closures, command checking and execution, procedure checking and bookkeeping, we need to introduce some new constructors besides those of Section 4.3.1 and new predicates besides $\Rightarrow$. As in [BH90], the use of these new constructors is reserved: a programmer cannot directly utilize these constructors to write down a program. Below we list the constructors and predicates, and we briefly describe the most important ones. For their formal meaning, see Theorem 4.3.

Constructor	Functionality
$[\_/\_]\_$	$: Expr \times Id \times Expr \rightarrow Expr$
$\_ \cdot \_$	$: Expr \times Expr \rightarrow Expr$
$[\_\|\_]\_$	$: Decl \times Commands \times Expr \rightarrow Expr$
$[\_/\_]_c\_$	$: Commands \times Id \times Commands \rightarrow Commands$
lambda	$: Id \times Id \times Commands \rightarrow Proc$
$[\_/\_]_{pe}\_$	$: Proc \times ProcId \times Expr \rightarrow Expr$
$[\_/\_]_{pc}\_$	$: Proc \times ProcId \times Commands \rightarrow Commands$

Judgement	Type	Judgement	Type
$\Rightarrow$	$\subset Expr \times Expr$	$\Rightarrow_p$	$\subset ProcId \times Proc$
$value$	$\subset Expr$	$free_e$	$\subset Expr \times IdSet$
$closed$	$\subset Expr$	$free_c$	$\subset Commands \times IdSet$
$closed_p$	$\subset ProcId$	$\triangleright$	$\subset Decl \times IdSet$

where *Proc* is a new syntactic class defined as $q ::= \mathbf{lambda}x, y.C$, and *IdSet* is the subset of *Expr* defined as $l ::= nil \mid x \mid l_1 :: l_2$

The intuitive meaning of $[n/x]M$ is "the expression obtained from M by replacing all free occurrences of x with n." Just as for the **let** discussed in Section 4.2.2, in order to evaluate $[n/x]M$ we have to evaluate M under the assumption that the value of x is n, and hence any previous assumption on x must be ignored. This is implemented by the substitution rule, A.1, which is similar to the **let** rule of Section 4.2.2. This rule is the core of the evaluation system. Many other evaluation rules, e.g. the one for **let**, are reduced to evaluation of a substitution (see e.g. the rule A.3). In NOS, to each sort of identifiers and

substitution operators (e.g. *Id* and $[\_/\_]\_$, *ProcId* and $[\_/\_]_{pe}\_$, *Id* and $[\_/\_]_{c}\_$, etc.) there corresponds a specific substitution rule, similar in shape to rule A.1. In fact, as we have seen in Section 4.2.2, bookkeepings can be fruitfully used whenever one has to deal with standard static scoping. One can even think of these rules as a polymorphic variant of the same set of rules. Of course, minor adjustments have to be accounted for (rules A.24, A.33, A.29).

The intuitive meaning of $[R|C]M$ is the same as of [**on** R **do** C]M. This expression is introduced in order to apply the declaration R until it is empty (rule A.16); then, the command C is executed.

The judgement *value* encodes the assertion that an expression is a value, and so it cannot be further reduced nor its meaning is affected by a substitution.

The judgements *free*, $free_c, \triangleright$ are used to check that command expressions, that is of the form [**on** D **do** C]M, and procedures do not access global variable. Informally, we can infer $\Gamma \vdash$ *free* M l if and only if all the free variables of M appear at the leaves of the tree l (see Theorem 4.3). On the other hand, $\triangleright$ collects the variables defined by a declaration D into a set (represented by a tree of identifiers).

The judgement $\Rightarrow_p$ is used for bookkeeping the bindings between procedure identifiers and procedural abstraction (see rules A.28, A.30).

The evaluation of λ-abstractions

It is worthwhile examining in detail how λ-abstractions are evaluated; this methodology is quite general, and it can be applied whenever one faces the evaluation of abstractions in a static-scoping language.

This mechanism makes use of an auxiliary judgement for each sort with variables, in order to determine the bindings that we have to record in the closure. In the case of $\mathcal{L}_P$, we introduce two judgements *closed*, $closed_p$. These judgements belong to static semantics: their derivation rules do not use evaluation rules. Informally, we can derive *closed* M if and only if M has no free variables; more precisely, $\Gamma \vdash$ *closed* M iff for every $x \in \mathrm{FV}(M)$, there is an assumption $(closed\ x) \in \Gamma$ (see Lemma 4.3 below).

Let us consider the rules for abstractions evaluation:

$$\frac{\begin{array}{c}(closed\ x)\\ \vdots \\ closed\ M\end{array}}{\textbf{lambda}\ x.M \Rightarrow \textbf{lambda}\ x.M}(A.4) \qquad \frac{\begin{array}{c}(closed\ y)\\ \vdots \\ y \Rightarrow n \quad \textbf{lambda}\ x.M \Rightarrow m\end{array}}{\textbf{lambda}\ x.M \Rightarrow [n/y]m}(A.5)$$

In order to apply rule (A.4), we need a *closed* assumption for each free variable of M but x. These assumptions are introduced only by the (A.5) rule, which at the same time extends the environment of the resulting closure by adding the corresponding binding (here, the constructor $[\_/\_]\_$ is used to record local environments). Therefore, an expression like **lambda** $x.M$ is evaluated into $[n_1/x_1]\ldots[n_k/x_k]$**lambda**$x.M$, where $x_1, \ldots, x_k$ are all the free identifiers of M but x, and $n_1, \ldots, n_k$ are their respective values. Application of closures should be self-explicative.

Notice that NOS is more efficient than SOS in building closures: bindings on variables which do not occurr free in **lambda** $x.M$ are not necessarily recorded. This does not hold for SOS, where a closure contains the whole definition environment [Plo81].

4.3.3 Adequacy

In this section we will show that the NOS description of $\mathcal{L}_P$ appearing in appendix A.1.1 is adequate w.r.t. the denotational semantics; that is, we will give soundness and completeness results of one semantics w.r.t. the other.

Definition 4.4 *A set of formulæ* Γ *is a* canonical hypothesis *if*

- *it contains only formulæ like "*$x \Rightarrow n, P \Rightarrow_p q, closed(x), closed_p(P)$*";*
- *if* $x \Rightarrow n, x \Rightarrow m \in \Gamma$ *then* m *and* n *are syntactically the same expression;*
- *if* $P \Rightarrow_p q, P \Rightarrow_p q' \in \Gamma$ *then* q *and* q' *are syntactically the same procedure;*

where $x \in Id, P \in ProcId$ *and* $m, n \in Expr, q, q' \in Proc$.

In the rest of section, Γ will denote a generic canonical hypothesis; therefore, we will not take into account "partial evaluations", that is assumptions of the form $M \Rightarrow m$ where M is not an identifier.

Definition 4.5 *Let* $M \in Expr, R \in Decl$ *and* Γ *be a canonical hypothesis; then*

1. *the set of* free identifiers of M *is denoted by* $\mathrm{FV}(M) \subset Id \cup ProcId$. FV *is naturally extended to Commands, bearing in mind that* $\mathrm{FV}(x := M) = \mathrm{FV}(M)$*;*
2. *the set of* variables defined by R *is* $\mathrm{DV}(M) \subset Id$*, defined as*

$$\mathrm{DV}(x_1 = M_1; \ldots; x_k = M_k) = \{x_1, \ldots, x_k\};$$

3. *the set of* Γ-closed identifiers $C(\Gamma)$ *is*

$$C(\Gamma) \stackrel{\text{def}}{=} \{x \in Id \mid closed(x) \in \Gamma\} \cup \{P \in ProcId \mid closed_p(P) \in \Gamma\};$$

4. *the* closure *of* Γ *is*

$$\overline{\Gamma} \stackrel{\text{def}}{=} \Gamma \cup \{closed(x) | (y \Rightarrow n) \in \Gamma, x \in \mathrm{FV}(n)\} \cup \{closed_p(P) | (y \Rightarrow n) \in \Gamma, P \in \mathrm{FV}(n)\};$$

5. Γ *is a* well-formed hypothesis, wfh, *if* $\Gamma = \overline{\Gamma}$.

Lemma 4.3 (Free Vars Lemma for NOS) $\forall \Gamma, \forall M, m \in Expr, \forall C \in Commands, \forall l \in IdSet$:

1. $\Gamma \vdash closed\ M \iff \mathrm{FV}(M) \subseteq C(\Gamma)$
2. $\Gamma \vdash free\ m\ l \iff \mathrm{FV}(m) \cap Id \subseteq leaves(l) \wedge \mathrm{FV}(m) \cap ProcId \subseteq C(\Gamma)$
   $\Gamma \vdash free\ C\ l \iff \mathrm{FV}(C) \cap Id \subseteq leaves(l) \wedge \mathrm{FV}(C) \cap ProcId \subseteq C(\Gamma)$
   where $leaves(nil) = \emptyset, leaves(x) = \{x\}, leaves(l_1 :: l_2) = leaves(l_1) \cup leaves(l_2)$
3. $\Gamma \vdash value\ m \implies \overline{\Gamma} \vdash closed\ m$

Proof. By induction on the derivations and on the syntax of M, m, C. □

Note that not all closed expressions are values; e.g., ((**lambda** $x.x$) 0).

Definition 4.6 *Let $I \subseteq Id \cup ProcId$ and let $\rho, \rho' \in Env$. We say that ρ and ρ'* agree on *I ($\rho \equiv_I \rho'$) if $\forall x \in I \cap Id : (access\ [\![x]\!]\ \rho = access\ [\![x]\!]\ \rho')$ and $\forall P \in I \cap ProcId : (procaccess\ [\![P]\!]\ \rho = procaccess\ [\![P]\!]\ \rho')$.*

Note that all ρ, ρ' agree on the empty set, that is $\forall \rho, \rho' \in Env : \rho \equiv_\emptyset \rho'$.

Proposition 4.4 $\forall m \in Expr, \forall R \in Decl, \forall \rho, \rho' \in Env$:

1. $\rho \equiv_{\mathrm{FV}(m)} \rho' \Rightarrow \mathcal{E}[\![m]\!]\rho = \mathcal{E}[\![m]\!]\rho'$
2. $\rho \equiv_{\mathrm{FV}(C)} \rho' \Rightarrow \mathcal{C}[\![C]\!]\rho = \mathcal{C}[\![C]\!]\rho'$
3. $\rho \equiv_{\mathrm{FV}(R)} \rho' \Rightarrow \mathcal{D}[\![R]\!]\rho = \mathcal{D}[\![R]\!]\rho'$

Proof. By simultaneous induction on the syntax of m, C, R. □

Proposition 4.5 $\forall \Gamma, \forall \rho, \rho' \in Env, \forall m \in Expr, \forall l \in IdSet$:

1. $\rho \equiv_{C(\Gamma)} \rho' \wedge \Gamma \vdash closed\ m \implies \mathcal{E}[\![m]\!]\rho = \mathcal{E}[\![m]\!]\rho'$
2. $\rho \equiv_{leaves(l)} \rho' \wedge \Gamma \vdash free\ C\ l \implies \mathcal{C}[\![C]\!]\rho = \mathcal{C}[\![C]\!]\rho'$
3. $\rho \equiv_{C(\Gamma)} \rho' \wedge \Gamma \vdash value\ m \implies \mathcal{E}[\![m]\!]\rho = \mathcal{E}[\![m]\!]\rho'$

Proof. Follows from Lemma 4.3 and Proposition 4.4. □

Corollary 4.6 $\Gamma \vdash value\ m \wedge C(\Gamma) = \emptyset \Longrightarrow \forall \rho, \rho' \in Env : \mathcal{E}[\![m]\!]\rho = \mathcal{E}[\![m]\!]\rho'$

Definition 4.7 *We say that $\rho \in Env$* satisfies *Γ ($\rho \models \Gamma$) if $\forall (x \Rightarrow n) \in \Gamma : access\ [\![x]\!]\ \rho = \mathcal{E}[\![n]\!]\rho$, and $\forall (P \Rightarrow q) \in \Gamma : procaccess\ [\![x]\!]\ \rho = \mathcal{Q}[\![q]\!]\rho$.*

This is another place where the conciseness of the N.D. formalism comes into play. The domain of environments satisfying a given Γ can be much larger than the set of variables which occur on the left of assumptions in Γ.

Theorem 4.7 $\forall M, m, \forall \Gamma\ wfh, \forall \rho : \rho \models \Gamma \wedge \Gamma \vdash M \Rightarrow m \implies \mathcal{E}[\![M]\!]\rho = \mathcal{E}[\![m]\!]\rho$

Proof. By induction on the structure of derivation, using the previous results. □

Corollary 4.8 (Soundness) $\forall M, m \in Expr : \emptyset \vdash M \Rightarrow m \implies \mathcal{E}[\![M]\!] = \mathcal{E}[\![m]\!]$.

Proof. Put $\Gamma = \emptyset$ in Theorem 4.7, and notice that $\emptyset$ is a wfh and $\forall \rho \in Env : \rho \models \emptyset$. □

Completeness. A completeness result is something like an "inverse" of Corollary 4.8. However, a literal converse of Corollary 4.8 cannot hold: for $M = m = (\mathbf{lambda}x.x0)$ it is $\mathcal{E}[\![M]\!] = \mathcal{E}[\![m]\!]$ but of course $\not\vdash M \Rightarrow m$. In fact, only some expressions can appear as values (see Lemma 4.3). We need a new definition:

Definition 4.8 *Let $M \in Expr$. An hypothesis Γ is* suitable for *M (written M-suit(Γ)), if $\forall x, P \in \mathrm{FV}(M) \exists (x \Rightarrow n), (P \Rightarrow_p q) \in \Gamma$ such that $\mathrm{FV}(n), \mathrm{FV}(q) \subseteq C(\Gamma)$.*

A hypothesis is suitable for M if it contains enough bindings to evaluate M. Therefore, we can state the completeness of Natural Operational Semantics with respect to Denotational Semantics as follows:

(Completeness of NOS wrt DS) $\forall M \in \mathit{Expr}, \forall \Gamma$ *well-formed hypothesis,* $\forall \rho \in \mathit{Env}$*: if* $\mathcal{E}[\![M]\!]\rho \neq \bot, \top$*, and* M-suit(Γ)*, and* $\rho \models \Gamma$*, then* $\exists m \in \mathit{Expr}$: $\Gamma \vdash M \Rightarrow m$.

The difficulty in the proof is this: given an expression M whose meaning, in a given environment, is a proper point of V, and a suitable hypothesis Γ, we have to build up a deduction $\Gamma \vdash M \Rightarrow m$, for some m.[3] This cannot be done by induction on the syntactic structure of M, since our language is higher order; in fact, the evaluation of M can use M itself, and not only its subterms (see e.g. rule A.21). Nevertheless, the theorem can be proved by using the technique of *inclusive predicates*, developed by Milner and Plotkin [Plo85, Gun92, Win93].

Adequacy w.r.t. the Structural Operational Semantics. An easier way for proving completeness of Natural Operational Semantics is to refer to a Structural Operational Semantics formulation of the semantics (*à la Plotkin*, [Plo81]). One can define a complete "input-output" SOS system for $\mathcal{L}_P$, that is a system as those described in Section 4.1. Such a system would deal with two evaluation judgements: *evaluation of expressions*, $\rho \vdash_{SOS} M \rightarrow m$, and *execution of commands*, $\rho \vdash_{SOS} C \rightarrow \rho'$ where ρ, ρ' are finite environments, i.e. they are defined on a finite number of identifiers, and m is a value. In the literature, there are many SOS specifications for languages similar to $\mathcal{L}_P$, which have been proved complete with respect to their denotational semantics [Plo81, Gun92, Win93].

It is an easy task prove that such a SOS specification is equivalent to the NOS one, that is, $\forall \rho$ finite environment, $\forall \Gamma, \forall M, m \in \mathit{Expr}$:

1. if $\forall (x \Rightarrow n) \in \Gamma : \rho(x) = n$ and $\Gamma \vdash M \Rightarrow m$, then $\rho \vdash_{SOS} M \rightarrow m$;
2. if $\rho \vdash_{SOS} M \rightarrow m$ and $\forall x \in \mathrm{FV}(M) : (x \Rightarrow \rho(x)) \in \Gamma$, then $\Gamma \vdash M \Rightarrow m$.

This is provable by induction on the derivations, using techniques similar to those adopted previously in this subsection. In particular, the completeness part (2) does not require the technique of inclusive predicates, but only a simpler structural induction on the derivation $\rho \vdash_{SOS} M \rightarrow m$.

4.4 Some remarks about language design

$\mathcal{L}_P$ is quite different from the language considered in [BH90]. There are several reasons for these changes. In some cases these are motivated by the desire to have a natural soundness result (see Section 4.3.1 for remarks concerning procedures).

Another difference is that in $\mathcal{L}_P$, commands are embedded into expressions by the **on-do** construct. A simpler formalism for applying directly commands to expressions is used in [BH90], i.e. the "modal" operator [ ] : *Commands* $\times$ *Expr* $\rightarrow$ *Expr*. Informally, the value of $[C]M$ is the value of M after the execution of C. The execution of C can affect any variable which is defined before its execution, but only locally to M: as any expression, the evaluation of $[C]M$ does not change the global environment any more than evaluating 0 or *nil*. We preferred to emphasize this point, by introducing the construct [**on** $x_1 = M_1; \ldots; x_k = M_k$ **do** $C]M$: here, we immediately know that only the "interface variables" $x_1 \ldots x_k$, whose scope is limited to C and M, are accessible by C.

[3] By Theorem 4.7, this m has the same meaning as M

4.5 Some extensions of $\mathcal{L}_P$

In this section we briefly describe some further extensions of $\mathcal{L}_P$ concerning complex declarations, structures and imperative modules. Their semantics can be expressed without stores because there is no variable sharing. See A.2 for their NOS and DS specifications. We deal with each extension by itself, by simply adding new rules to the formal system without altering the previous ones. This illustrates the modularity of NOS which allows us to add new rules for new constructs without changing the previous ones. For each extension, one can prove adequacy of NOS w.r.t. the denotational semantics, by simply discussing only the new cases due to the extra rules.

4.5.1 Complex Declarations

The language $\mathcal{L}_D$ is obtained from $\mathcal{L}_P$ by adding expressions of the form **let** R **in** M where R is a complex declaration like in Standard ML [MTH90]:

$$\begin{array}{lll} \mathit{Expr} & M ::= & \ldots \mid \textbf{let } R \textbf{ in } M \\ \mathit{Decl} & R ::= & \ldots \mid R; S \mid R \textbf{ and} S \end{array}$$

In spite of the syntactic simplicity of these extensions, it appears to be unavoidable to define an entire evaluation system for declarations (rules A.72–A.88). The value of complex declarations are finite sets of bindings, represented by expressions called *syntactic environments*; they are trees whose leaves are of the form $x \mapsto n$ where $\mapsto$: $\mathit{Id} \times \mathit{Expr} \to \mathit{Expr}$ is a new local constructor. We need to introduce furthermore several constructors and a judgement for applying such syntactic environments to expressions and declarations ($\{\_\}\_, \{\_\}_{d}\_$) and for inferring expression closures ($\langle\_\rangle\_, \gg$). Informally, one can derive $\Gamma \vdash R \gg I$ iff all expressions contained in R are closed in Γ and I is the set of identifiers defined by R. On the other hand, $\Gamma \vdash \mathit{closed}\ \langle I \rangle M$ iff all free variables in M but the ones in I are *closed* in Γ. Once the rules will be laid down, these fact will be formally provable. Using this set of rules, we can define precisely when a complex **let** is closed without using any evaluation, since *closed* is a property belonging to static semantics. An adequacy theorem similar to Theorem 4.3 can be proved for the system given in Section A.1.2. In [BH90] there is a simpler approach; it uses the complex declaration evaluation in order to determine the set of defined identifiers. This approach is not complete: there are closed expression whose *closed* property cannot be inferred in [BH90]'s system (e.g. **let** $o = (\textbf{lambda}\ x.xx); z = (oo)$ **in** z).

4.5.2 Structures and signatures

The language $\mathcal{L}_{M_F}$ (Figure 4.3) extends $\mathcal{L}_P$ by adding a module system like that of Standard ML [MTH90], where a module is "an environment turned into a manipulable object". Like SML, a module (here called *structure*) has a *signature*, and we can do *signature matching* in order to "cast" structures. However, there are some differences between SML and $\mathcal{L}_{M_F}$. First, in $\mathcal{L}_{M_F}$ structures and signatures are indeed expressions. Therefore, they may be associated to identifiers with simple **let**s, without using special constructs. These **let**s can appear anywhere in expressions, not only at top level. Structures and signatures can

$$
\begin{array}{rll}
\textit{LongId} & u ::= & x \mid u.x \\
\textit{Commands} & C ::= & \ldots \mid u := M \\
\textit{Expr} & M ::= & \ldots \mid \textbf{sig}\ x_1 \ldots x_k\ \textbf{end} \mid \textbf{struct}\ x_1 = M_1; \ldots; x_k = M_k\ \textbf{end} \\
 & & \mid M : N \mid \textbf{open}\ u\ \textbf{in}\ M
\end{array}
$$

Figure 4.3: $\mathcal{L}_{M_F}$, the extension for functional modules.

$$
\begin{array}{rll}
\textit{ModId} & T ::= & t_1 \mid t_2 \mid t_3 \mid \ldots \\
\textit{Expr} & M ::= & \ldots \mid T.f \\
\textit{Commands} & C ::= & \ldots \mid \textbf{module}\ T\ \textbf{is}\ x = M;\ \textbf{proc}\ P(y) = C; \textbf{func}\ f = N\ \textbf{in}\ D \\
 & & \mid T.P(M)
\end{array}
$$

Figure 4.4: $\mathcal{L}_{M_I}$, the extension for imperative modules.

be manipulated by common functions; however, there are not *functors* since the sharing specification is not implemented. The NOS should be self-explanatory.

4.5.3 Imperative modules (Abstract Data Types)

The extension $\mathcal{L}_{M_I}$ (see Figure 4.4) introduces modules *à la Morris* [Mor73]. In this formulation, a module is very close to an Abstract Data Type: it contains

1. a set of local variables, recording the *state* of the module; they are not accessible from outside the module;
2. some code for the initialization of the local variables above;
3. a set of procedures and functions which operate on these local variables and are the only part accessible from outside the module (the *interface*).

From outside a module we can only evaluate its functions, which do not produce side-effects, and execute its procedures, which can modify the state of the module (the value of local variables). In order to illustrate the idea, but w.l.o.g., we discuss only modules with exactly one local variable, one procedure with one argument (passed by value) and one function.

As for the previous languages, we do not need a representation of the store in defining the semantics of this kind of module [Don77]. The rules for the specification of the imperative modules are certainly the most complex of those discussed in this thesis. They are based on the principle of distributing as much as possible under the form of hypothetical assumption in deductions. In a module there are three informations: the state, the procedure and the function. Actually, only the state is subject to changes upon execution of the

module procedure. We split these three informations and record them using three different judgements (see rule A.114). The predicates of these assumptions are the following:

$$\begin{aligned} \Rightarrow_m &\subset ModId \times (Expr \times ModId) \\ \Rightarrow_{mp} &\subset (ModId \times ProcId) \times Q \\ \Rightarrow_{mf} &\subset (ModId \times Id) \times Expr \end{aligned}$$

We use a lot of syntactic sugar; for instance, we write $T.P \Rightarrow_{mp} \boldsymbol{\lambda} x, y.C$ instead of $\Rightarrow_{mp} ((T, P), \textbf{lambda}\ x, y.C)$.

When the state of a module changes (by executing its procedure), we have to substitute only the assumption involving $\Rightarrow_m$; the other two remain the same. Thus, while the procedure and the function are left associated to the original module identifier, the state becomes associated to a new *ModId*, and this substitution affects a part of the declaration to be evaluated (see rules A.116 A.115). The link between the new state and the procedures is maintained by the module identifier which appears on right of $\Rightarrow_m$ assumption: it is merely copied from the old assumption into the new one (rule A.116).

When a module procedure has to be executed ($T.P(M)$), first we look for the state of the module T, by requiring $T \Rightarrow_m (p, T')$. Here we find the original module identifier, T'. The invoked procedure is then associated to this identifier in the assumption $T'.P \Rightarrow_{mp}$ **lambda** $x, y.C$. After having bound x and y respectively to module variable value (p) and actual parameter (m), we execute C and get back the new value of the state variable. Finally, we substitute T with the new module state.

Function evaluation is similar to procedure call, but simpler (rule A.117).

4.6 Conclusions and Related Work

After its introduction [Kah87], Natural Semantics have been adopted by many researchers in specifying operational behaviours of programs; see e.g. [CDDK86, Des86, Ter95]. This formalism has been enriched by introducing *higher-order abstract syntax* [MP91, HP92, Han93]: the resulting paradigm, called *Extended Natural Semantics*, has been proved very suited for manipulating pure functional languages. In this chapter, we have seen how NS can be extended in order to deal also with imperative features.

We have seen that NOS handles successfully languages which do not allow variable aliasing, or sharing, We have shown some of these languages: functional languages extended with a restricted form of commands and procedures, blocks, complex declarations, modules *à la ML* (structures and signatures) and modules *à la Morris*. Exception handling should be added to $\mathcal{L}_{M_F}$ quite easily, along the line of [BH90].

Unfortunately, we are not able to express concisely the semantics of a truly imperative language using this formalism. It seems that one cannot represent simultaneously both the store and the environment by means of assumptions. Without encoding a store we cannot describe usual imperative phenomena like side-effects with aliasing, argument passage of parameters by-reference and so on. Therefore, this formalism seems not general enough to deal with expressions with side-effects, functions with local state variables or memoization, Pascal procedures (procedures with global variables and call-by-reference). For instance, we can try to model a bank account defining a function `withdraw` which takes the amount to be withdrawn (an example taken from [AS85]):

```
let cash = 100; withdraw = lambda a.[on bal=cash do bal:=bal-a]bal
   in let remaining = (withdraw 50)
      in (withdraw 30)
```

This program would be evaluated to 70 instead 20: the first withdraw has no effect. The reason is that in the closure of `withdraw`, `cash` is bound to 100, and this binding is reapplied to the local environment whenever `withdraw` is applied; this "reinitializes" `bal` to 100 each time (see rules A.5, A.6, A.8). Therefore, a `withdraw` cannot affect any following application.

We can successfully implement this bank account by using the imperative modules (Section 4.5.3), e.g. as follows:

```
module account is
  bal = 100;
  proc withdraw(amount) = bal := bal - amount;
  func balance = bal
in ...
```

Now we can withdraw an amount A by executing `account.withdraw(`A`)`, and know how much money we have left by evaluating `account.balance`.

However, even this notion of module is too weak to adequately model "functions with local state" as are necessary, for instance, in realizing memoized functions. In fact, as soon as an instance of a module is packaged within a λ-abstraction, its connection with its parent (definition) is severed.

The real lack of our languages with respect to Standard ML is the absence of the store: due to its imperative feature ML is a store-based language [MTH90]. Therefore, in order to capture fully the semantics of ML by a NOS specification we have to find some representation of the store.

Ideally, we would like to extend the formalism as much as is needed to describe the semantics of an untyped λ-calculus extended by primitives for manipulating side-effects, like ML's `ref`, `!` and `:=` [Har89, MTH90]. The NOS of this language should be easily extended to that of ML. The only solution, so far, is to represent stores by means of linear datastructures, such as lists of "location-value" pairs. These lists would be part of the evaluation judgement, which would become of the form $\langle M, \sigma\rangle \Rightarrow \langle m, \sigma'\rangle$. Therefore, these structures would be carried along the whole derivation, even if we never access to the store. This arises again the problematics we have discussed in Section 4.1.2.

Recently, some new approaches in the treatment of linear datatypes have been suggested by the application of Linear Logic [Gir87a]. Miller proposed a representation of imperative languages in Forum [Mil94]; however, its approach still embodies the whole store in every step of the derivations. A different, and more promising, approach is followed in the forthcoming *Linear Logical Framework* [Cer96].

Chapter 5

Modal Logics

Although Modal Logics are not properly program logics, many of their features have a bearing on the "modally flavoured" operators of many program logics, such as $\Box, \Diamond, \bigcirc$ of Temporal Logics [Sti92, Lam91, MP81], the "relativized" modal operators $[\cdot]\cdot, \langle\cdot\rangle\cdot$ of Propositional Dynamic Logic, First-Order Dynamic Logic, Hennessy-Milner logics [KT90, Har84, Har79, HM85], and even the μ, ν operators of μ-calculus [Koz83, Sti92] and Hoare triples [Hoa69, Apt81]. Hence, this study is preliminary to that of proper program logics.[1]

5.1 Syntax and Semantics

The formulæ of the basic modal propositional language Φ are defined by the following abstract syntax:

$$\Phi: \quad \varphi ::= p \mid \neg\varphi \mid \varphi \supset \psi \mid \Box\varphi$$

where p ranges over the set of *atomic proposition*, denoted by Φ_0. The constant $ff \in \Phi_0$ denotes the falsum. Given $\varphi \in \Phi$, we denote by $\mathrm{FV}(\varphi)$ the set of *(free) atomic predicate variables*, defined as usual; the notion of FV is extended to sets of formulæ: $\mathrm{FV}(\Gamma) = \cup_{\varphi\in\Gamma}\mathrm{FV}(\varphi)$. By $\varphi[x_1, \ldots, x_n]$ we denote a formula φ such that $\mathrm{FV}(\varphi) \subseteq \{x_1, \ldots, x_n\}$; we define $\Phi_X \stackrel{\mathrm{def}}{=} \{\varphi \in \Phi \mid \mathrm{FV}(\varphi) \subseteq X\}$. Finally, we take $\Diamond\varphi$ as a syntactic shorthand for $\neg\Box\neg\varphi$.

Semantics of modal logics is given on *Kripke structures* [HC84, vB83]. Let F range over *frames*, that is pairs $\langle S, R\rangle$ where S is a sets (the *states*), and $R \subseteq S \times S$ is a relation (the *accessibility relation*) between states. Let ρ range over *Env*, the environments (that is, functions $\Phi_0 \to \mathcal{P}(S)$). A *model* is a pair $\mathcal{M} = \langle F, \rho\rangle$; the interpretation of a formula φ in the model $\langle F, \rho\rangle$ is a set of states $[\![\varphi]\!]_F\rho \subseteq F$, defined by induction on the syntax of φ. Interpretations are extended to subsets of Φ:

$$\begin{array}{rcl} [\![\cdot]\!]_F & : & \Phi \to \mathit{Env} \to \mathcal{P}(S) \\ [\![p]\!]_F\rho & = & \rho(p) \\ [\![\neg\varphi]\!]_F\rho & = & S \setminus [\![\varphi]\!]_F\rho \\ [\![\varphi \wedge \psi]\!]_F\rho & = & [\![\varphi]\!]_F\rho \cap [\![\psi]\!]_F\rho \\ [\![\Box\varphi]\!]_F\rho & = & \{s \in S \mid \forall s' \in S. R(s, s') \Rightarrow s' \in [\![\varphi]\!]_F\rho\} \end{array}$$

[1] Some of the results of this chapter will appear in [AHMP97].

$$[\![\Gamma]\!]_F\rho \;=\; \bigcap_{\varphi\in\Gamma} [\![\varphi]\!]_F\rho$$

Given a formula φ, a model $\mathcal{M} = \langle F, \rho\rangle$ and a state s, we define when φ is true in s just as membership: $s \models_{\mathcal{M}} \varphi \iff s \in [\![\varphi]\!]_F\rho$. In particular, $s \models_{\mathcal{M}} \Box\varphi \iff \forall s'.s \to s' \Rightarrow s' \models_{\mathcal{M}} \varphi$. If φ is true in every state of a model $\mathcal{M}$, we say that φ *is valid in* $\mathcal{M}$ (written $\models_{\mathcal{M}} \varphi$). Hence, φ is valid in a model $\langle S, R, \rho\rangle$ iff $[\![\varphi]\!]_{\langle S,R\rangle}\rho = S$.

5.2 Consequence Relations

As we have already observed in Section 3.1, there are two main classes of Consequence Relation for Modal Logics, *truth* and *validity* CR's, accordingly to whether we require formulæ to hold in every state or not. Even more choices arises when we restrict our interpretation to only one model, or a class of structures. For definiteness, we recall here some definitions given in Section 3.1.

Definition 5.1 (Consequence Relations for Modal Logics) *Let $\mathcal{M}$ range over modal models, φ over formulæ and Γ over sets of formulæ.*

Truth CR's: *(a.k.a.* model local consequences *[vB83])*

- *the* truth CR with respect to $\mathcal{M} = \langle F, \rho\rangle$ *is*

  $$\Gamma \models_{\mathcal{M}} \varphi \iff \forall w.w \in [\![\Gamma]\!]_F\rho \Rightarrow w \in [\![\varphi]\!]_F\rho$$

  we say that φ is true in Γ with respect to $\mathcal{M}$ *if* $\Gamma \models_{\mathcal{M}} \varphi$*;*
- *let Λ be a set of models; the* truth CR with respect to Λ *is*

  $$\Gamma \models_{\Lambda} \varphi \iff \forall\mathcal{M} \in \Lambda.\Gamma \models_{\mathcal{M}} \varphi$$

  we say that φ is true in Γ with respect to Λ *if* $\Gamma \models_{\Lambda} \varphi$*;*
- *the* (absolute) truth CR *is* $\models \stackrel{\text{def}}{=} \bigcap_{\mathcal{M}} \models_{\mathcal{M}}$*, where $\mathcal{M}$ ranges over all modal models. We say that* φ is true in Γ *if* $\Gamma \models \varphi$*.*

Validity CR's: *(a.k.a.* model global consequences *[vB83])*

- *the* validity CR with respect to $\mathcal{M} = \langle F, \rho\rangle$ *is*

  $$\Gamma \models\!\!\models_{\mathcal{M}} \varphi \iff [\![\Gamma]\!]_F\rho = S \Rightarrow [\![\varphi]\!]_F\rho = S$$

  we say that φ is valid in Γ with respect to $\mathcal{M}$ *if* $\Gamma \models\!\!\models_{\mathcal{M}} \varphi$*;*
- *let Λ be a set of models; the* validity CR with respect to Λ *is*

  $$\Gamma \models\!\!\models_{\Lambda} \varphi \iff \forall\mathcal{M} \in \Lambda.\Gamma \models\!\!\models_{\mathcal{M}} \varphi$$

  we say that φ is valid in Γ with respect to Λ *if* $\Gamma \models\!\!\models_{\Lambda} \varphi$*;*
- *the* validity CR *is* $\models\!\!\models \stackrel{\text{def}}{=} \bigcap_{\mathcal{M}} \models\!\!\models_{\mathcal{M}}$*, where $\mathcal{M}$ ranges over all modal models. We say that* φ is valid in Γ *if* $\Gamma \models\!\!\models_{\mathcal{M}} \varphi$*.*

As an example of interesting class of structures, consider the set of frames whose accessibility relation is transitive, or reflexive, or both...

These CR's correspond to the *(model) global relation* and the *(model) local relation* of [vB83], respectively. They differ on the releavance given to assumptions: in the validity CR, formulæ of Γ are seen as *theorems*, true in every state, while in the truth CR they are *assumptions*, locally true in each state we consider. This difference is made apparant in the following result, where $\Box^0\varphi \stackrel{\text{def}}{=} \varphi$, $\Box^{n+1}\varphi \stackrel{\text{def}}{=} \Box\Box^n\varphi$:

Theorem 5.1 ([vB83]) *For $\Gamma \subseteq \Phi$, $\varphi \in \Phi$: $\Gamma \Vdash \varphi \iff \{\Box^n\psi \mid \psi \in \Gamma, n \in \mathcal{N}\} \models \varphi$.*

Moreover, the usual "deduction theorem" ("$\Gamma, \varphi \models \psi \iff \Gamma \models \varphi \supset \psi$") holds only for the true CR's: it is easy to see that $p \Vdash \Box p$, but of course $\not\Vdash p \supset \Box p$.

Beside these simple consequence relations for Modal Logics, it is useful to introduce a Multiple Consequence Relation. In fact, there are proof systems which represent simultaneously both truth and validity consequence, lead us to consider the following MCR in place of SCR.

Definition 5.2 (Multiple Consequence Relation for Modal Logics) *Let $\mathcal{M}$ range over modal models, φ over formulæ and Γ over sets of formulæ.*

- *the* MCR with respect to $\mathcal{M} = \langle F, \rho\rangle$ *is* $(\models^1_{\mathcal{M}}, \models^2_{\mathcal{M}})$ *where the two components* $\models^i_{\mathcal{M}} \subset \mathcal{P}(\Phi) \times \mathcal{P}(\Phi) \times \Phi$ *are defined as follows:*

$$\begin{array}{lll} \Gamma;\Delta \models^1_{\mathcal{M}} \varphi & \iff & (\forall s.s \models_{\mathcal{M}} \Gamma) \Rightarrow (\forall s.s \models_{\mathcal{M}} \Delta \Rightarrow s \models_{\mathcal{M}} \varphi) \\ \Gamma;\Delta \models^2_{\mathcal{M}} \varphi & \iff & (\forall s.s \models_{\mathcal{M}} \Gamma) \Rightarrow (\forall s.s \models_{\mathcal{M}} \varphi) \end{array}$$

- *let Λ be a set of models; the* MCR with respect to Λ *is composed by the following two relations:*

$$\Gamma;\Delta \models^i_{\Lambda} \varphi \iff \forall \mathcal{M} \in \Lambda.\Gamma;\Delta \models^i_{\mathcal{M}} \varphi$$

  for $i = 1, 2$.

- *the* (absolute) MCR *consists of the relations $\models^i \stackrel{\text{def}}{=} \bigcap_{\mathcal{M}} \models^i_{\mathcal{M}}$, ($i = 1, 2$) where $\mathcal{M}$ ranges over all modal models.*

It is clear that the simple consequence relations of Definition 5.1 can be defined as particular cases of the multiple consequence relation:

Proposition 5.2 *Let φ formula, and Γ, Δ sets of formulæ. Then, the following hold:*

$$\begin{array}{lll} \Delta \models_{\mathcal{M}} \varphi & \iff & \emptyset;\Delta \models^1_{\mathcal{M}} \varphi \\ \Gamma \Vdash_{\mathcal{M}} \varphi & \iff & \Gamma;\Delta \models^2_{\mathcal{M}} \varphi \end{array}$$

5.3 Proof Systems

In this section we present some proof systems, in Natural Deduction style, for Modal Logics. These systems will extend any known ND-style system for classical logic, such as the one presented in Figure 5.1.

$$\mathbf{N}C = \boxed{\begin{array}{ll} \Gamma, \varphi \vdash \varphi & \textsc{Raa}\ \dfrac{\Gamma, \neg\varphi \vdash \mathit{ff}}{\Gamma \vdash \varphi} \\ \supset\text{-I}\ \dfrac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \supset \psi} & \supset\text{-E}\ \dfrac{\Gamma \vdash \varphi \supset \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \psi} \\ \mathit{ff}\text{-I}\ \dfrac{\Gamma \vdash \varphi \quad \Gamma \vdash \neg\varphi}{\Gamma \vdash \mathit{ff}} & \mathit{ff}\text{-E}\ \dfrac{\Gamma \vdash \mathit{ff}}{\Gamma \vdash \varphi} \end{array}}$$

Figure 5.1: $\mathbf{N}C$, a minimal proof system for classical logic.

$$\mathbf{N}K = \mathbf{N}C + \boxed{\supset_\Box\text{-E}\ \frac{\Gamma \vdash \Box(\varphi \supset \psi) \quad \Gamma \vdash \Box\varphi}{\Gamma \vdash \Box\psi} \qquad \Box'\text{-I}\ \frac{\emptyset \vdash \varphi}{\emptyset \vdash \Box\varphi}}$$

Figure 5.2: $\mathbf{N}K$, the ND-style system for minimal modal logic.

5.3.1 Proof systems for truth

In Figure 5.2 we present $\mathbf{N}K$, a system for representing the truth consequence relation of modal logics. This system can be easily extended in order to represent stronger logics (such as KT, $K4$, KL, ...), by adding other rules corresponding to the usual axioms of Hilbert-style systems (Figure 5.3).

Notice that this system features a impure rule, namely $\Box'$-I, which is the usual necessitation rule for truth proof systems.

Proposition 5.3 *In* $\mathbf{N}K$*:* $\Gamma \vdash \varphi \iff \Gamma \models \varphi$

Proof. (Sketch) The rules are sound; moreover, the rules and axioms of a Hilbert-style systems for the truth for K (see e.g. [HC84]) are derivable. □

This result can be easily extended to the other systems.

Prawitz' systems for $S4$

In Figure 5.4 we recall two well-known systems for $S4$ of Prawitz', namely the "first" and the "third" version of [Pra65]. Notice that these systems are impure, due to the weird side-conditions on the $\Box$-introduction rules.

$$\boxed{\begin{array}{ll} \Box\text{-E}\ \dfrac{\Gamma \vdash \Box\varphi}{\Gamma \vdash \varphi} & \Box_\Box\text{-I}\ \dfrac{\Gamma \vdash \Box\varphi}{\Gamma \vdash \Box\Box\varphi} \\ \Box_\Diamond\text{-I}\ \dfrac{\Gamma \vdash \Diamond\varphi}{\Gamma \vdash \Box\Diamond\varphi} & \Box_\supset\text{-I}\ \dfrac{\Gamma \vdash \Box(\Box\varphi \supset \varphi)}{\Gamma \vdash \Box\varphi} \end{array}} \qquad \boxed{\begin{array}{ll} \mathbf{N}KT & = \mathbf{N}K + \Box\text{-E} \\ \mathbf{N}K4 & = \mathbf{N}K + \Box_\Box\text{-I} \\ \mathbf{N}KT4 & = \mathbf{N}KT + \Box_\Box\text{-I} \\ \mathbf{N}KT45 & = \mathbf{N}KT4 + \Box_\Diamond\text{-I} \\ \mathbf{N}KL & = \mathbf{N}K + \Box_\supset\text{-I} \end{array}}$$

Figure 5.3: ND-style rules and systems for various modal logics.

$$\begin{array}{rcl}
\mathbf{N}S4 & = & \mathbf{N}C + \boxed{\Box\text{-I}\ \dfrac{\Gamma \vdash \varphi}{\Gamma \vdash \Box\varphi}(*) \quad \Box\text{-E}\ \dfrac{\Gamma \vdash \Box\varphi}{\Gamma \vdash \varphi}} \\
\mathbf{N}S4_f & = & \mathbf{N}C + \boxed{\Box_f\text{-I}\ \dfrac{\Gamma \vdash \varphi}{\Gamma \vdash \Box\varphi}(**) \quad \Box\text{-E}\ \dfrac{\Gamma \vdash \Box\varphi}{\Gamma \vdash \varphi}} \\
(*) & = & \text{the formulæ in } \Gamma \text{ are boxed.} \\
(**) & = & \text{the proof of the premise has a fringe of modal formulæ.}
\end{array}$$

Figure 5.4: $\mathbf{N}S4$ and $\mathbf{N}S4_f$, two Prawitz' systems

$$\mathbf{N}K' = \mathbf{N}C + \boxed{\begin{array}{cc}
\Box''\text{-I}\ \dfrac{\Gamma \vdash \varphi}{\Gamma \Vdash \Box\varphi} & \supset''\text{-E}\ \dfrac{\Gamma \vdash \varphi \supset \psi \quad \Gamma \Vdash \varphi}{\Gamma \Vdash \psi} \\
\Box'''\text{-I}\ \dfrac{\Gamma \Vdash \varphi}{\Gamma \Vdash \Box\varphi} & \supset'''\text{-E}\ \dfrac{\Gamma \Vdash \varphi \supset \psi \quad \Gamma \vdash \varphi}{\Gamma \Vdash \psi} \\
\supset'\text{-E}\ \dfrac{\Gamma \Vdash \varphi \supset \psi \quad \Gamma \Vdash \varphi}{\Gamma \Vdash \psi} & \supset_\Box\text{-E}\ \dfrac{\Gamma \vdash \Box(\varphi \supset \psi) \quad \Gamma \vdash \Box\varphi}{\Gamma \vdash \Box\psi}
\end{array}}$$

$$\boxed{\begin{array}{rcl|rcl}
\mathbf{N}KT' & = & \mathbf{N}K' + \Box\text{-E} & \mathbf{N}K4' & = & \mathbf{N}K' + \Box_\Box\text{-I} \\
\mathbf{N}KT4' & = & \mathbf{N}KT' + \Box_\Box\text{-I} & \mathbf{N}KT45' & = & \mathbf{N}KT4' + \Box_\Diamond\text{-I} \\
\mathbf{N}KL' & = & \mathbf{N}K' + \Box_\supset\text{-I} & & &
\end{array}}$$

Figure 5.5: $\mathbf{N}K'$, the ND-style system for minimal validity modal logic, and its extensions.

Proposition 5.4 ([Pra65]) *The systems $\mathbf{N}S4$, $\mathbf{N}S4_f$ are sound and complete with respect to the class of transitive and reflexive models.*

5.3.2 Proof systems for validity

In Figure 5.5 we present $\mathbf{N}K'$, a system for representing the validity consequence relation of modal logics. This system can be easily extended in order to represent stronger logics (such as KT, $K4$, KL, ...), by adding other rules corresponding to the usual axioms of Hilbert-style systems.

Notice that this system deals with two consequence relations at once: the first one, $\vdash$, is defined by the sole classical system $\mathbf{N}C$ and derives the pure propositional tautologies. The other relation, $\Vdash$, represents the valid consequences of the minimal modal logic, K. It is worthwhile noticing that the system $\mathbf{N}K'$ is pure, that is none of its rules has side-conditions.

Proposition 5.5 *In $\mathbf{N}K'$: $\Gamma \Vdash \varphi \iff \Gamma \Vdash\!\!\models \varphi$*

Proof. (Sketch) Rules are sound; moreover, the rules and axioms of an Hilbert-style systems for the validity for K (see e.g. [HC84]) are derivable. □

This result can be easily extended to the other systems.

$$\mathbf{N}K'' \stackrel{\text{def}}{=} \mathbf{N}K' + \left[\begin{array}{lll} \Gamma, \varphi \Vdash \varphi & \textsc{Embed}' \ \dfrac{\vdash \varphi}{\Vdash \varphi} & \textsc{Raa}_T \ \dfrac{\Gamma, \neg\varphi \Vdash \mathit{ff}}{\Gamma \Vdash \varphi} \\ \supset_T\text{-I} \ \dfrac{\Gamma, \varphi \Vdash \psi}{\Gamma \Vdash \varphi \supset \psi} & & \supset_T\text{-E} \ \dfrac{\Gamma \Vdash \varphi \supset \psi \quad \Gamma \Vdash \varphi}{\Gamma \Vdash \psi} \\ \mathit{ff}_T\text{-I} \ \dfrac{\Gamma \Vdash \varphi \quad \Gamma \Vdash \neg\varphi}{\Gamma \Vdash \mathit{ff}} & & \mathit{ff}_T\text{-E} \ \dfrac{\Gamma \Vdash \mathit{ff}}{\Gamma \Vdash \varphi} \end{array}\right]$$

$\mathbf{N}KT''$	$=$	$\mathbf{N}K'' + \Box$-E	$\mathbf{N}K4''$	$=$	$\mathbf{N}K'' + \Box_\Box$-I
$\mathbf{N}KT4''$	$=$	$\mathbf{N}KT'' + \Box_\Box$-I	$\mathbf{N}KT45''$	$=$	$\mathbf{N}KT4'' + \Box_\Diamond$-I
$\mathbf{N}KL''$	$=$	$\mathbf{N}K'' + \Box_\supset$-I			

Figure 5.6: $\mathbf{N}K''$, the ND-style system for minimal truth-validity modal logic, and its extensions.

5.3.3 A proof system for both truth and validity

We can introduce ND-style systems for truth consequences, based on the multiple CR ND-style system $\mathbf{N}K$' for validity. We need only to add a third consequence relation, namely $\Vdash$, with exactly the same rules as $\vdash$, and in addition the rule Embed'. The whole system, $\mathbf{N}K''$, is shown in Figure 5.6.

This system deals with three consequence relations at once: as for $\mathbf{N}K'$, $\vdash$ derives the pure propositional tautologies, while $\Vdash$ represents the valid consequences of the minimal modal logic, K. The new relation $\Vdash$ derives instead the true consequences of the minimal modal logic. It is worthwhile noticing that the system $\mathbf{N}K''$ is pure, that is none of its rules has side-conditions.

It is therefore natural to see this system as a representation of both the truth and the validity consequence relation; therefore, we give the following adequacy result with respect to the multiple consequence relation of Definition 5.2:

Proposition 5.6 *In* $\mathbf{N}K''$*, we have*

$$\begin{array}{rcl} \emptyset; \Delta \models^1 \varphi & \iff & \Gamma \Vdash \varphi \\ \Gamma; \Delta \models^1 \varphi & \iff & \Gamma \Vdash \bigwedge \Delta \supset \varphi \end{array}$$

Proof. (Sketch) Rules are sound; moreover, the rules and axioms of an Hilbert-style systems for the validity for K (see e.g. [HC84]) are derivable at the level of $\Vdash$. Completeness follows from the fact that $\varphi_1, \ldots, \varphi_n \models \varphi$ iff $\varphi_1 \supset \ldots \supset \varphi_n \supset \varphi$ is valid, and $\Vdash \varphi_1 \supset \ldots \supset \varphi_n \supset \varphi$. By applying the Embed' rule, we obtain $\Vdash \varphi_1 \supset \ldots \supset \varphi_n \supset \varphi$; the thesis follows by n applications of rule $\supset_T$-I. □

This system can be easily extended in order to represent stronger logics (such as KT, $K4$, KL, ...), by adding other rules corresponding to the usual axioms of Hilbert-style systems.

Chapter 6

Propositional Dynamic Logic

Propositional Dynamic Logic is one of the first modal logics for dealing with programs and processes. The adjective "propositional" is due to the fact that in *PDL* one focus on the structure of programs, by abstracting on the particular effect of atomic actions.

6.1 Syntax and Semantics

Let Act be a set of atomic command symbols, ranged over by a; the language $\mathcal{L}_{PDL}$ of Propositional Dynamic Logic is obtained by extending a propositional language with Kleene's algebra of *regular programs* $Prog$ on Act and the modal formula constructor $[\cdot]\cdot$ (Figure 6.1). Here, p, q range over atomic propositional symbols, and a ranges over atomic command symbols. The construct $\langle\cdot\rangle\cdot$ is defined as a shorthand for $\neg[\cdot]\neg\cdot$. The informal meaning of some of these constructs is the following:

$b?$ = "test b; proceed if true, fail if false"

c^* = "execute c a nondeterministically chosen finite number of times"

$[c]\varphi$ = "at the end of every non-diverging execution of c, φ holds"

$\langle c\rangle\varphi$ = "there is an execution of c such that at its end φ holds"

Despite its simplicity, the class of regular programs is very expressive; for instance, the usual **while** language, and many other constructs such as Dijkstra's guarded commands, can be defined in terms of regular programs [KT90, p.794]:

$$\begin{aligned}
\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 &\stackrel{\text{def}}{=} (b?; c_1) + (\neg b?; c_2)\\
\textbf{while } b \textbf{ do } c &\stackrel{\text{def}}{=} (b?; c)^*; \neg b?\\
\textbf{do } b_1 \to c_1 \;[]\ldots[]\; b_n \to c_n \textbf{ od} &\stackrel{\text{def}}{=} (b_1?; c_1 + \ldots + b_n?; c_n)^*; (\neg b_1 \wedge \ldots \wedge \neg b_n)?
\end{aligned}$$

The formal interpretation of *PDL* comes from Modal Logic. A *Kripke model* for *PDL* is a pair $\mathcal{M} = \langle S, [\![\cdot]\!]_{\mathcal{M}}\rangle$ where S is a (generic) set of *abstract states* (the *carrier*), ranged over by s and $[\![\cdot]\!]_{\mathcal{M}}$ is the interpretation of atomic propositional and command symbols: for all

Command-free Formulæ	$B:$	$b ::= q \mid b \supset b \mid b \wedge b \mid \neg b$
Regular Programs	$Prog:$	$c ::= a \mid b? \mid c; c \mid c + c \mid c^*$
Formulæ	$\Phi:$	$\varphi ::= p \mid \varphi \supset \psi \mid \varphi \wedge \psi \mid \neg\varphi \mid [c]\,\psi$

Figure 6.1: $\mathcal{L}_{PDL}$, the language of Propositional Dynamic Logic.

p, a, we have $[\![p]\!]_{\mathcal{M}} \subset S$ and $[\![a]\!]_{\mathcal{M}} : S \to \mathcal{P}(S)$.[1] The interpretation is then compositionally extended to all formulæ and programs:

$$\begin{array}{llll} [\![\varphi \supset \psi]\!] & \stackrel{\text{def}}{=} & (S \setminus [\![\varphi]\!]) \cup [\![\psi]\!] & [\![\varphi \wedge \psi]\!] \stackrel{\text{def}}{=} [\![\varphi]\!] \cap [\![\psi]\!] \\ [\![\neg\varphi]\!] & \stackrel{\text{def}}{=} & S \setminus [\![\varphi]\!] & [\![[c]\,\varphi]\!] \stackrel{\text{def}}{=} \{s \in S \mid [\![c]\!]s \subseteq [\![\varphi]\!]\} \\ [\![b?]\!] & \stackrel{\text{def}}{=} & \lambda s.[\![b]\!] \cap \{s\} & [\![c_1; c_2]\!] \stackrel{\text{def}}{=} [\![c_2]\!] \circ [\![c_1]\!] \\ [\![c_1 + c_2]\!] & \stackrel{\text{def}}{=} & [\![c_1]\!] \cup [\![c_2]\!] & [\![c^*]\!] \stackrel{\text{def}}{=} [\![c]\!]^* = \lambda s. \cup_{n\in\omega} [\![c]\!]^n s \end{array}$$

6.2 Consequence Relations

In this section, we introduce some consequence relations for *PDL*. We will see that, beside the usual truth and validity CR's, another distinction arise in connection with *finiteness* of assumptions. In fact, we are naturally lead to consider infinitary proof systems. Hence we will consider both a finitary and a infinitary proof system for *PDL*.

Definition 6.1 (Consequence Relations for *PDL*) *Let $\mathcal{M}$ be a model for PDL and $[\![\cdot]\!]_{\mathcal{M}}$ be the relative interpretation in $\mathcal{M}$ of atomic propositional symbols, extended compositionally to all formulæ and commands. The* truth *and the* validity CR's *for PDL wrt $\mathcal{M}$ are two relations $\models_{\mathcal{M}}, \Vdash_{\mathcal{M}} \subseteq \mathcal{P}(\Phi) \times \Phi$, defined as follows:*

$$\begin{array}{lcl} \Gamma \models_{\mathcal{M}} \varphi & \iff & [\![\Gamma]\!]_{\mathcal{M}} \subseteq [\![\varphi]\!]_{\mathcal{M}} \\ \Gamma \Vdash_{\mathcal{M}} \varphi & \iff & [\![\Gamma]\!]_{\mathcal{M}} = S \Rightarrow [\![\varphi]\!]_{\mathcal{M}} = S \end{array}$$

The (absolute) truth *and* validity CR's *for PDL are defined as follows:*

$$\begin{array}{lcl} \Gamma \models \varphi & \iff & \forall \mathcal{M}.\Gamma \models_{\mathcal{M}} \varphi \\ \Gamma \Vdash \varphi & \iff & \forall \mathcal{M}.\Gamma \Vdash_{\mathcal{M}} \varphi \end{array}$$

The finitary *truth consequence relations with respect to a model $\mathcal{M}$ is the restriction of $\models_{\mathcal{M}}$ to finite sets:*

$$\begin{array}{lcl} \Gamma \models_{f\,\mathcal{M}} \varphi & \iff & \exists \Delta \subseteq \Gamma,\ \textit{finite}.\Delta \models_{\mathcal{M}} \varphi \\ \Gamma \models_{f} \varphi & \iff & \exists \Delta \subseteq \Gamma,\ \textit{finite}.\Delta \models \varphi \end{array}$$

It is well-known that *PDL* lacks compactness. In fact, the CR's $\models, \Vdash, \models_{\mathcal{M}}, \Vdash_{\mathcal{M}}$ above may involve infinite sets of assumptions, where all assumptions are essential:

[1] These models are also known as *transition systems*; see for instance [Sti92].

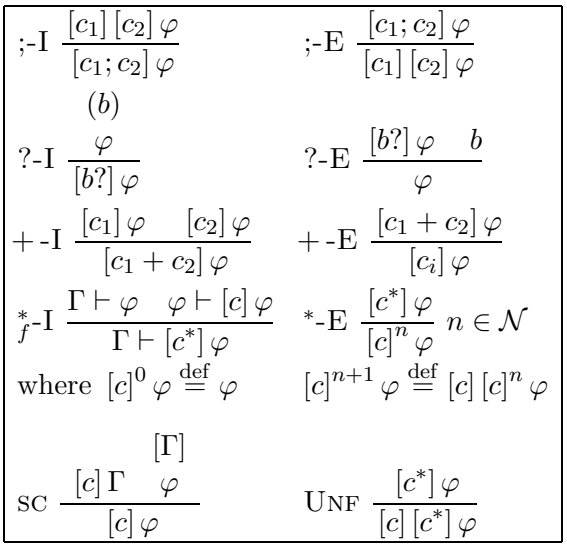

Figure 6.2: The modal rules of the system $\mathbf{N}_f PDL$.

Proposition 6.1 *There is an infinite set* Γ, φ *formula such that* $\Gamma \models \varphi$, *but for all* $\Delta \subseteq \Gamma$, Δ *finite:* $\Delta \not\models \varphi$.

Proof. See [Har84, Theorem 2.15]. Take e.g. $\{[x := x\dot{-}1]^n\, x \neq 0 \mid n \in \mathcal{N}\} \models [(x := x\dot{-}1)^*]\, x \neq 0$. □

If we are interested in sound finitary system for reasoning about $\models$, we can content ourselves with just having completeness wrt $\models_f$,

6.3 Proof Systems

We present both a finitary proof system, complete with respect to the finitary truth CR, and an infinitary proof system, which we prove complete with respect to the infinitary consequence relation $\models$.

6.3.1 A finitary ND-style system for *PDL*

In Figure 6.2 we introduce a finitary proof systems for *PDL*. This system has a strong modal flavour. The rule SC is the Natural Deduction-style version of Scott's rule; moreover, the rule $^*_f$-I is submitted to a "proof condition" on the right subderivation. Indeed, the same rule could be written as follows:

$$\frac{\Gamma \vdash \varphi \quad \emptyset \vdash \varphi \supset [c]\,\varphi}{\Gamma \vdash [c^*]\,\varphi}$$

which makes the condition more explicit.

$\mathbf{N}_f PDL$ is a sound and complete representation of $\models_f$.

Proposition 6.2 $\forall \Gamma$ *finite,* φ *formula:* $\Gamma \vdash_{\mathbf{N}_f PDL} \varphi \iff \Gamma \models_f \varphi$.

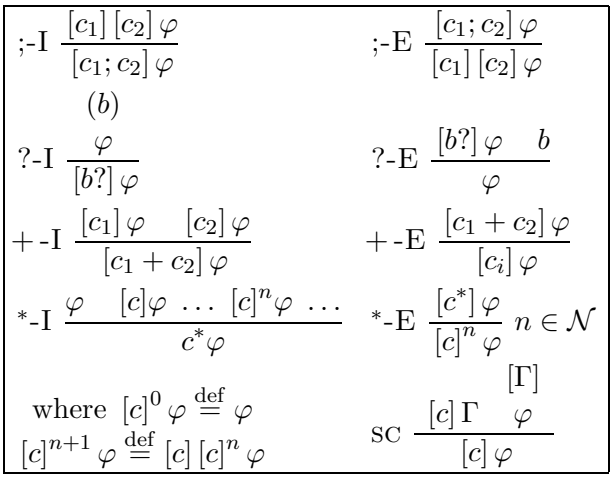

Figure 6.3: The modal rules of the system $\mathbf{N}PDL$.

Proof. (Sketch) The rules are sound; on the other hand, it is easy to prove that the axioms and rules of a complete Hilbert-style proof system for *PDL* (see e.g. [Har84, KT90]) are derivable. □

However, due to well-known compactness problems, a finitary proof system cannot be complete with respect to the whole relation $\models$:

Proposition 6.3 $\mathbf{N}_f PDL$ *is incomplete with respect to* $\models$.

Proof. This is a standard compactness argument. Let $\Gamma \models \varphi$ the true consequence of Proposition 6.1, and suppose that $\Gamma \vdash_{\mathbf{N}_f PDL} \varphi$. Since any proof in $\mathbf{N}_f PDL$ can take into account only a finite set of assumptions, there is a finite set $\Delta \subseteq \Gamma$ such that $\Delta \vdash_{\mathbf{N}_f PDL} \varphi$. By the Proposition, we know that $\Delta \not\models \varphi$, hence $\mathbf{N}_f PDL$ is unsound—a contradiction. □

For this reason, in the next section we will examine an infinitary proof system which will overcome this problem.

6.3.2 An infinitary ND-style system for *PDL*

We introduce a ND-style *infinitary system* for the truth CR, namely $\mathbf{N}PDL$, which we prove to be sound and complete wrt $\models$. The system $\mathbf{N}PDL$ is obtained by adding the rules appearing in Figure 6.3 to the usual ND-style system for propositional logic [Pra65], such as $\mathbf{N}C$ (Figure 5.1). Notice that the rule *-I is infinitary; in sequent form it appears as follows:

$$^*\text{-I}\ \frac{\{\Gamma \vdash [c]^n\,\varphi \mid n \in \mathcal{N}\}}{\Gamma \vdash [c^*]\,\varphi}.$$

Despite there is no more the rule $^*_f$-I, the system $\mathbf{N}PDL$ has to be accounted for a modal system, due to the presence of the characteristic SC rule.

In the system $\mathbf{N}PDL$ we have *exactly one* introduction rule for each program constructor and the elimination rules are induced by the former. Moreover, there is no need of the "spurious" unfolding rule UNF.

N*PDL* is a sound and strongly complete representation of $\models$:

Theorem 6.4 (Infinitary Completeness) $\forall \Gamma, \varphi : \Gamma \vdash_{\mathbf{N}PDL} \varphi \iff \Gamma \models \varphi$.

Proof. (Hint) Soundness is straightforward; just prove that each rule is sound. Strong completeness can be shown by generalizing the proof given by Harel [Har84, Theorem 3.15] for the (finitary) completeness of the Hilbert-style calculus for *PDL*. The basic idea is to follow the proof of Fischer-Ladner, by adapting suitably the notion of subformula and by using, as atoms, infinite sets of formulæ instead of finite conjunctions. A similar (and more complex) proof will be carried out for First-Order Dynamic Logic; see Theorem 7.4. □

6.4 Proof Theory

Consider the following "reduced" modality rule:

$$\textsc{sc}_0 \ \frac{[a]\,\Gamma \quad \overset{[\Gamma]}{\varphi}}{[a]\,\varphi} \quad \text{that is} \quad \textsc{sc}_0 \ \frac{\{\Delta \vdash \Box a\psi \mid \psi \in \Gamma\} \quad \Gamma \vdash \varphi}{\Delta \vdash [a]\,\varphi} a \in Act$$

It is just the SC rule, restricted to atomic commands. Despite this restriction, this rule is enough for deriving the more general SC rule.

Theorem 6.5 *The rule* SC *is admissible in the system* $(\mathbf{N}PDL - \textsc{sc}) + \textsc{sc}_0$.

Proof. (Sketch) Let $\Gamma \vdash \varphi$; then we have to prove that for all $c \in Prog$, we prove also $[c]\Gamma \vdash [c]\varphi$. This can be proved by induction on the syntax of the command c. □

This proof have been carried out formally in the system Coq.

Chapter 7

First-Order Dynamic Logic

Dynamic Logic has been thoroughly investigated from the model theoretic point of view. Not as much attention, however, has been payed to its proof theory or to the possibility of representing consequence relations different from that of *validity*. The relevant concept being that of *theoremhood*, the proof systems considered have been mainly Hilbert-style systems [Har79, Har84, KT90, Sti92]. There is only one remarkable exception, albeit unpublished, of ND-style System for Deterministic *DL* due to C. Stirling [Sti85] (see Section 7.6).[1]

7.1 Syntax and Semantics

First-Order Dynamic Logic differs from Propositional Dynamic Logic because we choose *assignments* as atomic commands. The language $\mathcal{L}_{DL}$ is defined by extending any first-order language by a new formula constructor "$[\cdot]\cdot$" and by introducing the new syntactic domains, of *command-free formulæ* and of *regular programs*, as appears in Figure 7.1 (where, for definiteness, we focus on the first-order language of Peano Arithmetic, $\mathcal{L}_{PA}$).

The semantics of *DL* is given by extending the interpretation of first-order logic to the domain *Prog*. In the style of Denotational Semantics, we define the interpretation of terms and formulæ by choosing a model $\mathcal{M}$ for the underlying first-order logic and defining a (polymorphic) interpretation function $[\![\cdot]\!]_{\mathcal{M}}$ for terms and formulæ:

$$\mathcal{M} \stackrel{\text{def}}{=} \langle D, 0, 1, +, *, < \rangle \qquad \mathit{Env}_{\mathcal{M}} \stackrel{\text{def}}{=} \mathit{Var} \to D$$
$$[\![\cdot]\!]_{\mathcal{M}} : \mathit{Term} \to \mathit{Env}_{\mathcal{M}} \to D \qquad [\![\cdot]\!]_{\mathcal{M}} : \Phi \to \mathcal{P}(\mathit{Env}_{\mathcal{M}})$$

$\mathit{Env}_{\mathcal{M}}$ is the domain of *environments* (or *states, assignments*) and it is ranged over by s, s_1, s_2. The meaning of a term is a function from environments to elements in the domain D; the meaning of a formula is the set of environments which satisfy it. These two functions are compositionally defined on the syntax of phrases in the obvious way; the extra case is dealt by taking $[\![[c]\, p]\!]_{\mathcal{M}} \stackrel{\text{def}}{=} \{s \in \mathit{Env}_{\mathcal{M}} \mid [\![c]\!]_{\mathcal{M}} s \subseteq [\![p]\!]_{\mathcal{M}}\}$.

The function $[\![\cdot]\!]_{\mathcal{M}} : \mathit{Prog} \to \mathit{Env}_{\mathcal{M}} \to \mathcal{P}(\mathit{Env}_{\mathcal{M}})$ arises from the interpretation of *PDL* (Section 6.1), as soon as we choose the set of environments *Env* as carrier of the dynamic

[1] Some of the results of this chapter appeared also in [HM96].

Identifiers	*Var* :	$x ::= i_0 \mid i_1 \mid i_2 \mid i_3 \mid \ldots$
Terms	*Term* :	$t ::= 0 \mid 1 \mid x \mid t + t \mid t * t$
Propositional Formulæ	B :	$b ::= t = t \mid t < t \mid b \supset b \mid b \wedge b \mid \neg b$
Command-free Formulæ	*Form* :	$\varphi ::= t = t \mid t < t \mid \varphi \supset \varphi \mid \varphi \wedge \varphi \mid \neg\varphi \mid \forall x\varphi$
Regular Programs	*Prog* :	$c ::= x := t \mid b? \mid c; c \mid c + c \mid c^*$
Formulæ	Φ :	$\varphi ::= t = t \mid t < t \mid \varphi \supset \varphi \mid \varphi \wedge \varphi \mid \neg\varphi \mid \forall x\varphi \mid [c]\,\varphi$

Figure 7.1: $\mathcal{L}_{DL}$, the language of First-Order Dynamic Logic (on Peano Arithmetic).

model. Hence, the semantics of programs is as follows:

$$\begin{array}{rcl@{\qquad}rcl} [\![x := t]\!]_{\mathcal{M}} & \stackrel{\text{def}}{=} & \lambda s.\{s[x \mapsto [\![t]\!]_{\mathcal{M}} s]\} & [\![c_1; c_2]\!] & \stackrel{\text{def}}{=} & [\![c_2]\!] \circ [\![c_1]\!] \\ [\![c^*]\!] & \stackrel{\text{def}}{=} & \lambda s. \cup_{n\in\omega} [\![c]\!]^n s & [\![b?]\!] & \stackrel{\text{def}}{=} & \lambda s.[\![b]\!] \cap \{s\} \\ [\![c_1 + c_2]\!] & \stackrel{\text{def}}{=} & [\![c_1]\!] \cup [\![c_2]\!] & & & \end{array}$$

where the composition operator is extended to sets in the obvious way.

An alternative way for defining the semantics of regular programs is to give *relations* between states in place of functions from states to sets of states. That is, we can define $[\![\cdot]\!]_{\mathcal{M}} : \mathit{Prog} \to \mathcal{P}(\mathit{Env}_{\mathcal{M}} \times \mathit{Env}_{\mathcal{M}})$ as well. It is clear that the two approaches are equivalent; we preferred the functional one because it is closer in spirit to the denotational paradigm. Nevertheless, viewing commands as relations between states is customary in the logical tradition (see e.g. [KT90, Har84]); therefore, we will denote by $[\![c]\!]_{\mathcal{M}}$ both the function of arity $\mathit{Env}_{\mathcal{M}} \to \mathcal{P}(\mathit{Env}_{\mathcal{M}})$ and the equivalent relation, subset of $\mathit{Env}_{\mathcal{M}} \times \mathit{Env}_{\mathcal{M}}$.

7.2 Consequence Relations

Besides absolute validity and absolute truth, various CR's can be introduced according to the class of models that one focuses on. Since in this chapter we focus on the language of Peano Arithmetic (PA), we consider two classes of models: the class of *all* first-order structures which are models of *PA*, and that consisting only of the standard model (denoted by $\mathcal{N}$). Truth CR's for *DL* are defined by suitably specializing the general definition of consequence relations on first-order structures (Definition 3.4).

Definition 7.1 *Let $\mathcal{L} = \langle T, \Phi\rangle$ a first-order language and $[\![\cdot]\!]_{\mathcal{M}}$ its interpretation, as above. We define*

- *the* truth CR *with respect to $\mathcal{M}$ is*

$$\Gamma \models_{\mathcal{M}} \varphi \iff (\forall \rho.\rho \in [\![\Gamma]\!]_{\mathcal{M}} \Rightarrow \rho \in [\![\varphi]\!]_{\mathcal{M}})$$

- *let Λ be a set of first-order structures for L; the* truth CR *with respect to Λ is*

$$\Gamma \models_{\Lambda} \varphi \iff (\forall \mathcal{M} \in \Lambda \forall \rho.\rho \in [\![\Gamma]\!]_{\mathcal{M}} \Rightarrow \rho \in [\![\varphi]\!]_{\mathcal{M}})$$

- *the* truth CR *is $\models \stackrel{\text{def}}{=} \bigcap_{\mathcal{M}} \models_{\mathcal{M}}$, where $\mathcal{M}$ ranges over all the first-order structures (of the right signature); in other words,*

$$\Gamma \models \varphi \iff \forall \mathcal{M} \forall \rho.\rho \in [\![\Gamma]\!]_{\mathcal{M}} \Rightarrow \rho \in [\![\varphi]\!]_{\mathcal{M}}$$

System	Composition	Finitary	Features	Completeness
$\mathbf{N}_f DL$	$\mathbf{N}FOL$+$\mathbf{N}_f PDL$ +Figure 7.3	Yes	no symmetry, proof rules	wrt termination assertions
$\mathbf{N}_f^a DL$	$\mathbf{N}_f DL$+CONVER	Yes	no symmetry, proof rules	arithmetical
$\mathbf{N}DL$	$\mathbf{N}FOL$+$\mathbf{N}PDL$ +Figure 7.3	No	symmetry	absolute
$\mathbf{N}DL$+Δ	$\mathbf{N}DL$+Δ	No	symmetry, proper axioms	wrt models of Δ
$\mathbf{N}^a DL$	$\mathbf{N}DL$+Δ_{PA}+ CONVER	No	symmetry, proper axioms, proof-rules	wrt the only $\mathcal{N}$

Figure 7.2: A summary of Proof Systems for Dynamic Logic.

The correctness consequences *with respect to a model $\mathcal{M}$ are all the true consequences of the form $\Gamma \models_{\mathcal{M}} [c]\,\varphi$, for Γ, φ command and quantifier free. The* termination consequences *with respect to a model $\mathcal{M}$ are all the true consequences of the form $\Gamma \models_{\mathcal{M}} \langle c\rangle\,\varphi$, for Γ, φ command and quantifier free.*

7.3 Proof systems

In this section we will give

- a finitary ND-style proof system for the uninterpreted Dynamic Logic, that is a system complete w.r.t. the termination assertions of *DL*;
- a finitary ND-style proof system for the interpreted Dynamic Logic, that is a proof system *arithmetically* complete w.r.t. the finite $\models$ (See [KT90, Har84] or Section 7.3.1 for definitions).
- an infinitary ND-style proof system for the uninterpreted Dynamic Logic, that is a proof system complete w.r.t. the whole $\models$.
- an infinitary ND-style proof system for the Dynamic Logic interpreted on naturals, that is a proof system complete w.r.t. the finite $\models_{\mathcal{N}}$.

The whole situation is summarized in Figure 7.2.

7.3.1 Finitary ND-style systems for *DL*

In this section we present $\mathbf{N}_f DL$, a finitary ND-style system for Dynamic Logic. This system is obtained by adding to the usual ND-style system for Peano Arithmetic [Pra65] the finitary rules for Propositional Dynamic Logic (Figure 6.2) and the rules in Figure 7.3.

The system $\mathbf{N}_f DL$ is indeed a ND-style system, since there are introduction rules for each program constructor and the corresponding elimination rules are induced by the introduction rules. The rules for equality and the quantifier are more involved than the

$$:=\text{-I}\ \frac{\Gamma, y = t \vdash \varphi[y/x]}{\Gamma \vdash [x := t]\,\varphi}\ y \notin \mathrm{FV}(\Gamma, \varphi, t)$$

$$:=\text{-E}\ \frac{\Gamma_1 \vdash [x := t]\,\varphi \quad \Gamma_2, \varphi[y/x], y = t \vdash \psi}{\Gamma_1, \Gamma_2 \vdash \psi}\ y \notin \mathrm{FV}(\Gamma_2, \varphi, \psi, t)$$

$$\forall\text{-I}\ \frac{\Gamma \vdash \varphi}{\Gamma \vdash \forall x \varphi}\ x \notin \mathrm{FV}(\Gamma)$$

$$\forall\text{-E}\ \frac{\Gamma_1 \vdash \forall x \varphi \quad \Gamma_2, \varphi[y/x], y = t \vdash \psi}{\Gamma_1, \Gamma_2 \vdash \psi}\ y \notin \mathrm{FV}(\Gamma_2, \forall x \varphi, t, \psi)$$

$$\textsc{CongrId}\ \frac{\Gamma_1 \vdash \varphi \quad \Gamma_2 \vdash x = y}{\Gamma_1, \Gamma_2 \vdash \varphi[y/x]}\ y \notin \mathrm{FV}(\varphi)$$

$$\textsc{Congr}\ \frac{\Gamma_1 \vdash \varphi[t_1/x] \quad \Gamma_2 \vdash t_1 = t_2}{\Gamma_1, \Gamma_2 \vdash \varphi[t_2/x]}\ \varphi \text{ is command-free}$$

Figure 7.3: The specific rules for the system $\mathbf{N}_f DL$.

usual ones for FOL, due to the presence of commands. Reflexivity of equality can be encoded immediately, but the usual rules of congruence, i.e.

$$\frac{\varphi[t_1/x] \quad t_1 = t_2}{\varphi[t_2/x]}$$

have to be rephrased with care: derivations like

$$\frac{[x := 0]\,(x = 0) \qquad x = 1}{[x := 0]\,(1 = 0)}$$

have to be prevented. To this end, we introduce two rules: Congr and CongrId. Congr can be applied only to command-free formulæ, i.e. formulæ where no command appears. CongrId, can be applied to any formula, since it merely replaces all occurrences of an identifier with a new identifier.

The non traditional form of $\forall$-elimination is due to the fact that, in general, the quantified formula φ may contain commands, and therefore not all occurrences of a bound variable can be replaced by a term. For instance, $\forall x.\,[x := 0]\,(x = 0)$ holds, but its naïve instantiation $[1 := 0]\,(1 = 0)$ is clearly meaningless. A correct formulation of instantiation of quantified variables is in fact one of the most difficult technical issues to deal with in encoding *DL*. As we have already seen in Section 3.2.3, in Hilbert systems this is usually achieved by replacing, whenever required, any program c with the equivalent "normal form" $z_1 := x_1; \ldots; z_n := x_n; c'; x_1 := z_1; \ldots; x_n := z_n$ where the x_i's are all the identifiers appearing in c, the z_i's are fresh and c' is obtained from c by replacing the x_i's with z_i's (see [Har84]). This solution is clearly cumbersome if we want to use practically the formal system. The problematic nature of instantiation of quantifiers lies, as in the case of the congruence rules, in the different nature of pure logical identifiers and program variables. In fact, the property "$s \in [\![\varphi[t/x]]\!] \iff s[x \mapsto [\![t]\!]s] \in [\![\varphi]\!]$" does not hold for *DL*.

Instead, the following holds:

Lemma 7.1 (Substitution Lemma for *DL*) *For all models $\mathcal{M}$:*

$$\textsc{Conver}\ \frac{\emptyset \vdash p[x+1/x] \supset \langle c \rangle\, p \quad \Gamma \vdash p[t/x]}{\Gamma \vdash \langle c^* \rangle\, p[0/x]}\ x \notin \mathrm{FV}(c)$$

$$\textsc{Induc}\ \frac{\emptyset \vdash [c]\, p \supset p[x+1/x] \quad \Gamma \vdash [c^*]\, p[0/x]}{\Gamma \vdash p[t/x]}\ x \notin \mathrm{FV}(c)$$

Figure 7.4: The convergence and induction rules.

1. $\forall x \in \mathit{Var}, \forall t, u \in \mathit{Term}, \forall s \in \mathit{Env}_{\mathcal{M}} : [\![t[u/x]]\!]s = [\![t]\!](s[x \mapsto [\![u]\!]s])$
2. $\forall x \in \mathit{Var}, \forall t \in \mathit{Term}, \forall b \in B, \forall s \in \mathit{Env}_{\mathcal{M}} : s \in [\![b[t/x]]\!] \iff s[x \mapsto s(t)] \in [\![b]\!]$
3. $\forall x, y \in \mathit{Var}, \forall \varphi \in \Phi, \forall s \in \mathit{Env}_{\mathcal{M}} : s \in [\![\varphi[y/x]]\!] \iff s[x \mapsto s(y)] \in [\![\varphi]\!]$

Our solution to the instantiation problem is to replace the bound variable x with a fresh variable y, and to assume $y = t$ in the minor premise. The usual $\forall$-elimination rule is derivable in the case of command-free predicates.

The system $\mathbf{N}DL$ is sound and complete with respect to the termination consequences:

Theorem 7.2 *For all Γ finite set of command-free formulæ, φ command-free formula, c command:*

$$\Gamma \vdash_{\mathbf{N}_f DL} \langle c \rangle\, \varphi \iff \Gamma \models \langle c \rangle\, \varphi.$$

Proof. (Sketch) The rules are sound; on the other hand, it is easy to prove that the axioms and rules of a complete Hilbert-style proof system for termination assertions of DL (see e.g. [Har84, Section 3.4.1]) are derivable. □

Arithmetical Completeness

It is well known that, even for programming languages less expressive than regular programs, there is no complete finitary axiomatization of their Hoare or Dynamic Logic with respect to a sufficiently expressive model. The usual approach adopted in these case is to "translate" formulæ of DL into (semantically) equivalent f.o. formulæ, and then to add an *oracle* for the first-order theory of a specific model. This translation is however possible only if the specific model has enough expressive power. These models, called *arithmetical* (also called *acceptable*, *expressive*) [Har84, KT90, Apt81], include (a first-order definable copy of) $\mathcal{N}$, and first-order definable gödelization functions for the encoding/decoding of arbitrary finite sequences of elements of the domain, into (the copy of) $\mathbf{N}$. For instance, $\mathcal{N}$ and most datastructures arising in computer science (lists, trees,...) are arithmetical.

Like for Hilbert-style systems, we can obtain an arithmetically complete proof system $\mathbf{N}_f^a DL$ by adding either the *convergence rule* or the equivalent dual *induction principle rule* (Figure 7.4). Both these rules are "impure" in the sense of Avron [Avr91], and are proof-rules, since the first premise has to be a theorem.

Let us denote by $\mathbf{N}_f^a DL$ the system obtained by adding either Conver or Induc to $\mathbf{N}_f DL$. This system is still a finitary system. Like for Hilbert-style systems, we can introduce an infinitary degree of power by adding an *oracle* for the first-order theory of a specific model. Let $\mathrm{Th}(\mathcal{M})$ be the set of first-order theorems of a model $\mathcal{M}$ (in general,

this set is not r.e.), and denote by $\Gamma, \mathrm{Th}(\mathcal{M}) \vdash_{\mathbf{N}^a_f DL} \varphi$ the derivability of φ in $\mathbf{N}^a_f DL$ from the assumptions Γ and the set $\mathrm{Th}(\mathcal{M})$.

Theorem 7.3 (Arithmetical Completeness) *Let Γ be a set of formulæ, φ a formula, $\mathcal{M}$ a model; then,*

$$\Gamma, \mathrm{Th}(\mathcal{M}) \vdash_{\mathbf{N}^a_f DL} \varphi \iff \Gamma \models_{f\,\mathcal{M}} \varphi$$

Proof. (Sketch) The axioms and rules of an arithmetical complete Hilbert-style proof system for *DL* (see e.g. [Har84, Section 3.5.2]) are derivable. □

7.3.2 Infinitary ND-style systems for *DL*

In order to represent the infinitary $\models$, in this section we present $\mathbf{N}DL$, an infinitary ND-style system for Dynamic Logic.

The basic system $\mathbf{N}DL$ is obtained by adding to the usual ND-style system $\mathbf{N}FOL$ for First-Order Logic [Pra65] the infinitary rules for Propositional Dynamic Logic (Figure 6.3) and the rules in Figure 7.3 for assignment, quantification and congruence.

This system can be extended by adding specific axioms for a specific class of models. For instance, let Δ_{PA} be the set of axioms of Peano Arithmetic; then $\mathbf{N}DL + \Delta_{PA}$ is the system obtained by adding Δ_{PA} as proper axioms to $\mathbf{N}DL$. For Δ a set of f.o. formulæ, let $\Lambda(\Delta)$ be the class of f.o. models for Δ, that is $\Lambda(\Delta) \stackrel{\text{def}}{=} \{\mathcal{M} \mid \forall \varphi \in \Delta.\ \models_{\mathcal{M}} \varphi\}$. Then, the following main theorem holds.

Theorem 7.4 (Infinitary Completeness) *Let Γ a set of formulæ, φ a formula, Δ a set of f.o. formulæ. Then,*

$$\Gamma \vdash_{\mathbf{N}DL+\Delta} \varphi \iff \Gamma \models_{\Lambda(\Delta)} \varphi.$$

Proof. Soundness is easy. Completeness is proved by modifying suitably the proof of Theorem 3.15 in [Har84]. This is a standard Henkin argument, adapted to the peculiarities of Dynamic Logic. Let $\Gamma \nvdash_{\mathbf{N}DL+\Delta} \varphi$; then, $\Gamma, \Delta, \neg\varphi$ is consistent, that is, $\Gamma, \Delta, \neg\varphi \nvdash_{\mathbf{N}DL} ff$. By the following Model Existence Lemma (Lemma 7.5), there is a model $\mathcal{M}$ which satisfies $\Gamma, \Delta, \neg\varphi$, which is equivalent to say that $\Gamma, \Delta \not\models_{\mathcal{M}} \varphi$; hence, *a fortiori*, $\Gamma \not\models_{\mathcal{M}} \varphi$. Since $\mathcal{M}$ satisfies Δ, it is that $\mathcal{M} \in \Lambda(\Delta)$, from which we have the thesis. □

Lemma 7.5 (Model Existence Lemma) *For A set of formulæ, if $A \nvdash ff$ then A is satisfiable.*

Proof. See Appendix B.1. □

Corollary 7.6 *Let Γ a set of formulæ, φ a formula. Then*

$$\Gamma \vdash_{\mathbf{N}DL} \varphi \iff \Gamma \models \varphi$$
$$\Gamma \vdash_{\mathbf{N}DL+\Delta_{PA}} \varphi \iff \Gamma \models_{PA} \varphi$$

where $\models_{PA}$ is the consequence relation on the class of models for Peano Arithmetic.

Notice that the completeness result in the case of Peano Arithmetic is different from the arithmetical completeness of finitary proof systems.

Focusing on consequences true in all models of Peano Arithmetic, the system $\mathbf{N}DL + \Delta_{PA}$ rules out many interesting consequences which are true when reasoning about real programs which utilize as datatype the real integers. For instance, the formula $\varphi \stackrel{\text{def}}{=} \langle (x := x - 1)^* \rangle (x = 0)$ is not valid: take any nonstandard model $\mathcal{N}^*$, and consider the state s such that $s(x) = \nu$, ν a nonstandard integer; then, $s \notin [\![\varphi]\!]_{\mathcal{N}^*}$. The same happens with the **while**-termination formula $\langle \textbf{while } x > 0 \textbf{ do } x := x - 1 \rangle (x = 0)$. This is the reason for focusing on the sole standard model of arithmetic and the associated CR $\models_{\mathcal{N}}$.

In order to represent $\models_{\mathcal{N}}$, we extend the system $\mathbf{N}DL + \Delta_{PA}$ to the new system $\mathbf{N}^a DL$, by adding either the convergence rule or the equivalent dual induction principle rule (Figure 7.4). That is, we define

$$\mathbf{N}^a DL \stackrel{\text{def}}{=} \mathbf{N}DL + \Delta_{PA} + \textsc{Conver}$$

One can easily see that

$$\vdash_{\mathbf{N}^a DL} \langle (x := x - 1)^* \rangle (x = 0)$$
$$\vdash_{\mathbf{N}^a DL} \langle \textbf{while } x > 0 \textbf{ do } x := x - 1 \rangle (x = 0)$$

It is interesting to notice that the rule *-I is enough to recover the full power of the ω-rule of infinitary first order logic:

Theorem 7.7 *Let p be any command-free formula; then, the ω-rule*

$$\frac{\{\Gamma \vdash \varphi[n/x] \mid n \in \omega\}}{\Gamma \vdash \forall x \varphi}$$

is derivable in $\mathbf{N}^a DL$.

Proof. (Sketch) The proof relies upon the fact that command iteration is nondeterministic, hence $\forall x \varphi$ is equivalent to $y = 0 \supset [y := y + 1^*] \varphi[y/x]$ (y fresh). Each premise $\varphi[n/x]$ in the ω-rule can be rendered by means of the formula $y = 0 \supset [y := y + 1]^n \varphi[y/x]$ (y fresh); applications of *-I and Induc yield the ω-rule. □

$\mathbf{N}^a DL$ is sound and complete with respect to the standard model of integers.

Theorem 7.8 $\forall \Gamma, \varphi : \Gamma \vdash_{\mathbf{N}^a DL} \varphi \iff \Gamma \models_{\mathcal{N}} \varphi$.

Proof. Follows from Theorems 7.4, 7.7. □

Here again, this form of completeness is different from the arithmetical one of finitary proof systems.

7.4 Proof Theory

Proposition 7.9 *The usual $\forall$-elimination rule of first order logic,*

$$\frac{\Gamma \vdash \forall x \varphi}{\Gamma \vdash \varphi[t/x]} \varphi \text{ command free}$$

is derivable in $\mathbf{N}_f DL$, $\mathbf{N}DL$.

Proof. Consider the following derivation of the ∀-elimination rule:

$$\dfrac{\forall x\varphi \qquad \dfrac{(\varphi[y/x])_1 \quad (y=t)_1}{\varphi[t/x]}\dagger}{p[t/x]}(1)$$

where † is a sound application of CONGR since p is command-free. □

Proposition 7.10 *The useful, albeit "impure", $[\cdot]$-intro rule*

$$\frac{\emptyset \vdash \varphi}{\emptyset \vdash [c]\,\varphi}$$

and the rule SC *are admissible for* $\mathbf{N}DL$

Proof. (Sketch) By induction on the proofs. □

Instead of introducing proof rules, we could have used alternatively *non-interference* judgements *à la* Reynolds [Rey78] as side conditions of the rules. These are judgements which generalize side-conditions such as $x \notin \mathrm{FV}(A)$, allowing for a finer control of interference between formulæ and commands. A deeper discussion on the use of non-interference judgements in proof systems for program logics can be found in Section 8.3.

7.5 Equivalence of programs

An interesting application of the proof systems for Dynamic Logic (and hence of the proof editor generated from the signature $\Sigma(DL)$ using Coq, Section 14.2) is the possibility of proving formally the equivalences of programs, in the style of Meyer and Halpern [MH82].

Definition 7.2 (Equivalence of programs) *Two programs $c, d \in Prog$ are said to be* equivalent ($[\![c]\!] = [\![d]\!]$) *iff* $\forall \mathcal{M} : [\![c]\!]_{\mathcal{M}} = [\![d]\!]_{\mathcal{M}}$.

In other words, two programs are equivalent iff there is no model in which they can be distinguished one from the other. The encodings of $\mathbf{N}DL$ could be particularly suited for *computer-assisted* proofs of equivalence of programs, based on higher-order logic, since they naturally provide metalogical facilities such as quantifications on predicates (i.e. second-order quantifications) and proofs by induction on the structure of predicates.

In the following, we denote by *CClauses* and *DClauses* the following sets of *disjunctive* and *conjunctive clauses*:

$$\begin{aligned} DClauses &\stackrel{\text{def}}{=} \{x_1 \neq y_1 \vee \ldots \vee x_n \neq y_n \mid n \in \mathcal{N}\} \\ CClauses &\stackrel{\text{def}}{=} \{x_1 = y_1 \wedge \ldots \wedge x_n = y_n \mid n \in \mathcal{N}\} \end{aligned}$$

The following result is an adaptation of [MH82, Lemma 4.3].

Lemma 7.11 *Let $c_1, c_2 \in Prog$ be two programs and $\mathcal{M}$ a model such that $[\![c_1]\!]_{\mathcal{M}} \setminus [\![c_2]\!]_{\mathcal{M}} \neq \emptyset$. Then, there is $\varphi \in DClauses$ such that $[c_2]\,\varphi \not\models_{\mathcal{M}} [c_1]\,\varphi$*

Proof. Let $X \stackrel{\text{def}}{=} \mathrm{FV}(c_1, c_2) = \{x_1, \ldots, x_n\}$ the set of variables occurring in c_1, c_2, and let $(s, r) \in [\![c_1]\!]_{\mathcal{M}} \setminus [\![c_2]\!]_{\mathcal{M}}$; this is equivalent to

$$\forall r' \in [\![c_2]\!]_{\mathcal{M}} s : \exists x_m \in X . r'(x_m) \neq r(x_m). \tag{7.1}$$

Let then $X' \stackrel{\text{def}}{=} \{x'_1, \ldots, x'_n\}$ be n new variables; we define then an environment s' and a formula φ as follows:

$$s'(x) \stackrel{\text{def}}{=} \begin{cases} s(x_i) & \text{if } x \equiv x_i \text{ for some } x_i \in X \\ r(x_i) & \text{if } x \equiv x'_i \text{ for some } x'_i \in X' \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\varphi \stackrel{\text{def}}{=} x_1 \neq x'_1 \vee \ldots \vee x_n \neq x'_n$$

We show that $s' \models_{\mathcal{M}} [c_2]\,\varphi$. Let $r' \in [\![c_2]\!]_{\mathcal{M}} s'$; we prove that $r' \models \varphi$. Since s and s' take the same value on X, by (7.1) there is $x_m \in X$ such that $r'(x_m) \neq r(x_m)$. By definition, we know that $s'(x'_m) = r(x_m)$, and $s'(x'_m) = r'(x'_m)$ because $x'_m \notin X$ and hence it cannot be modified by c_2. We have thence $r'(x_m) \neq r'(x'_m)$, so $r' \models x_m \neq x'_m$ and then $r' \models \varphi$.

We show now that $s' \not\models_{\mathcal{M}} [c_1]\,\varphi$, that is, there is $r' \in [\![c_1]\!]_{\mathcal{M}} s'$ such that $\forall i . r'(x_i) = r'(x'_i)$. Let us define r' as follows:

$$r'(x) \stackrel{\text{def}}{=} \begin{cases} r(x_i) & \text{if } x \equiv x_i \text{ for some } x_i \in X \\ s'(x) & \text{otherwhise} \end{cases}$$

Clearly, $r' \in [\![c_1]\!]_{\mathcal{M}} s'$ because $r \in [\![c_1]\!]_{\mathcal{M}} s$ (that is, $(s, r) \in [\![c_1]\!]_{\mathcal{M}}$) and on the variables which are not modified by c_1, r' is equivalent to s'. Then, $r'(x_i) = r(x_i) = s'(x'_i) = r'(x'_i)$, hence $\forall i . r'(x_i) = r'(x'_i)$, which is the thesis. □

Proposition 7.12 *Let c_1, c_2 be programs; then, the following are equivalent:*

1. $[\![c_1]\!] = [\![c_2]\!]$;
2. $\forall \varphi \in \mathit{DClauses} : [c_1]\,\varphi \models [c_2]\,\varphi$ *and* $[c_2]\,\varphi \models [c_1]\,\varphi$.

Proof. $(1 \Rightarrow 2)$: trivial.
$(2 \Rightarrow 1)$: By contraposition: if $[\![c_1]\!] \neq [\![c_2]\!]$, then there is a model $\mathcal{M}$ such that $[\![c_1]\!]_{\mathcal{M}} \setminus [\![c_2]\!]_{\mathcal{M}} \neq \emptyset$ or $[\![c_2]\!]_{\mathcal{M}} \setminus [\![c_1]\!]_{\mathcal{M}} \neq \emptyset$; then, by Lemma 7.11, there is $\varphi \in \mathit{DClauses}$ such that $[c_2]\,\varphi \not\models [c_1]\,\varphi$ or $[c_1]\,\varphi \not\models [c_2]\,\varphi$. □

Theorem 7.13 $\forall c, d$ *programs, the following are equivalent:*

1. $[\![c]\!] = [\![d]\!]$
2. $\forall \varphi \in \mathit{DClauses} : [c]\,\varphi \vdash_{\mathbf{N}_f DL} [d]\,\varphi$ *and* $[d]\,\varphi \vdash_{\mathbf{N}_f DL} [c]\,\varphi$;
3. $\forall \varphi \in \mathit{CClauses} : \langle c \rangle\,\varphi \vdash_{\mathbf{N}_f DL} \langle d \rangle\,\varphi$ *and* $\langle d \rangle\,\varphi \vdash_{\mathbf{N}_f DL} \langle c \rangle\,\varphi$;

Proof. $(1 \iff 2)$: by Proposition 7.12 and finitary completeness of $\mathbf{N}_f DL$ (Th. 7.2).
$(2 \iff 3)$: by propositional reasoning, the following equivalence is easily proved

$$\begin{aligned} [c]\,(x_1 \neq x'_1 \vee \ldots x_n \neq x'_n) &\equiv [c]\,\neg(x_1 = x'_1 \wedge \ldots x_n \wedge x'_n) \\ &\equiv \neg\,\langle c \rangle\,(x_1 = x'_1 \wedge \ldots x_n \wedge x'_n) \end{aligned}$$

and by propositional completeness, $\neg\,\langle c_2 \rangle\,\varphi \vdash \neg\,\langle c_1 \rangle\,\varphi$ iff $\langle c_1 \rangle\,\varphi \vdash \langle c_2 \rangle\,\varphi$. □

Remark 1. The result holds also for the infinitary system $\mathbf{N}DL$, of course, but the finitary version suffices because we never need an infinite set of assumptions. This is not surprising, because only a finite number of variables can appear in a regular command, and hence we need a finitary amount of information for distinguish two regular programs.

Remark 2. The comparision of Theorem 7.13 with the corresponding [MH82, Theorem 6.1, Corollary 6.2] is insightful of the extra power provided by a system which permits the elimination of boxed fomulæ. Meyer and Halpern focused on Hilbert-style systems for Hoare-like assertions as a tool for deriving "observations" on the programs, that is pairs "precondition-postcondition". In fact, the following is a consequence of Lemma 7.11:

Corollary 7.14 *Let $c_1, c_2 \in Prog$ be two programs and $\mathcal{M}$ a model such that $[\![c_1]\!]_{\mathcal{M}} \setminus [\![c_2]\!]_{\mathcal{M}} \neq \emptyset$. Then, there are two first order formulæ φ, ψ such that $\models \varphi \supset \langle c_1 \rangle \psi$ and $\not\models \varphi \supset \langle c_2 \rangle \psi$.*

The proof of this Lemma relies on the existence of a f.o. formula, namely φ, which represents the input-output relation of c_1 and whose construction is not trivial—it involves a gödelization of the formula $[c_1]\psi$, possibly extending the model with a copy of the ring of integers (see [MH82, Lemma 4.2]). This complication is due to the fact that in Hoare-like logics most of the logical manipulation is delegated to the underlying first-order calculus (see e.g. Cook's completeness, or arithmetical completeness [Har79]). This drawback is solved by adopting a finitary complete system, where we can deal directly with formulæ such as $[c_1]\psi$, on a par with first-order formulæ; in fact, the consequence $[c_1]\psi \models [c_2]\psi$ is a shorter form for $\models \varphi \supset [c_2]\psi$, where φ is the weakest precondition of c_1, ψ. The use of Natural Deduction style system $\mathbf{N}_f DL$, moreover, allows us for using "elimination" rules for commands, differently from Hilbert-style systems.

7.6 Related Work

Our approach was inspired by the unpublished handwritten notes [Sti85] by Colin Stirling. A ND-style system for Deterministic Dynamic Logic is sketched there. The fundamental idea of Stirling's approach is to "divorce the notion of free occurrence of a variable from that of substitution". The system deals with assertions of the form $p\theta$, where θ is called *(explicit) substitution*: $\theta ::= \varepsilon \mid ({}^{t}_{x}\theta)$. A phrase of the form ${}^{t_1}_{x_1} \ldots {}^{t_n}_{x_n}$ represents a sequence of "delayed" substitutions. Substitutions are not performed until the formula on which they are applied is box-free. This idea is inspiring but it is clearly impractical. $\mathbf{N}DL$ retains this, while overcomes the explicit substitution problem in the assignment rules, by taking full advantage of assumptions, i.e. distributing the substitution in the proof context (see rules := -I, := -E). Of course, this is sound only with respect to the truth consequence relation.

The technique of treating substitutions by means of sets of assumptions can be seen as a particular case of the bookkeeping approach, described in Section 4.2.2.

Chapter 8

Hoare Logic

As for Dynamic Logic, in most of the researches about Hoare Logic very little attention has been payed to the underlying consequence relations [Apt81, AO91, Cou90, LS88]. In this chapter, we introduce and compare truth and validity CR's which arise over Hoare triples applying Definition 3.4 to *HL*(Section 8.2). In Section 8.3, we will introduce a family of Natural Deduction-style proof systems for Hoare Logic; we will prove their adequacy with respect to the truth CR's. In Section 8.4, we will see that Natural Deduction style systems for truth Hoare Logic are strictly tied to proof systems for Dynamic Logics; indeed, a proof system for *HL* will be derived in $\mathbf{N}DL$.

8.1 Syntax and Semantics

A syntax for Hoare Logic is defined starting from a given first-order language with equality (without quantifiers), by adding constructors for a nondeterministic **while** language and Hoare triples [Apt81]. Here we focus on an extension of the language for (quantifier free) Peano Arithmetic, PA^{qf}. We introduce the syntactic classes of *nondeterministic **while** programs*, *Hoare triples* and *assertions* listed in Figure 8.1. Let φ range over f.o. formulæ, c over $Prog$; h over HT; a over Φ; Γ, Δ range over finite sets of formulæ Φ.

Each new syntactic class is interpreted by extending the semantic function $[\![\cdot]\!]$:

$$[\![\cdot]\!]_{\mathcal{M}} : Prog \to Env_{\mathcal{M}} \to \mathcal{P}(Env_{\mathcal{M}}) \qquad [\![\cdot]\!]_{\mathcal{M}} : HT \to \mathcal{P}(Env_{\mathcal{M}}) \qquad [\![\cdot]\!]_{\mathcal{M}} : \Phi \to \mathcal{P}(Env_{\mathcal{M}})$$

Since we deal with nondeterministic programs, more than one environment can be the result of a command execution. The meaning of programs can be defined in several ways, depending on the kind of semantics we want to examine. Denotational models can be used as well as operational ones: in the latter case one defines $[\![c]\!]s \stackrel{\mathrm{def}}{=} \{s' \mid \langle c, s\rangle \stackrel{*}{\to} \langle \varepsilon, s'\rangle\}$ where $\stackrel{*}{\to}$ is the transitive closure of the transition relation defined by the operational model. We do not insist on the definition of $[\![\cdot]\!]_{\mathcal{M}}$ — see e.g. [Sch86, Sti92].

Hoare triples are interpreted as usual: $[\![\{\varphi\}c\{\psi\}]\!]_{\mathcal{M}} \stackrel{\mathrm{def}}{=} \{s \in Env_{\mathcal{M}} \mid s \in [\![\varphi]\!]_{\mathcal{M}} \Rightarrow [\![c]\!]_{\mathcal{M}} s \subseteq [\![\psi]\!]_{\mathcal{M}}\}$. This definition reflects the partial correctness of Hoare Logic. A Hoare triple is always satisfied by an environment which leads the command to diverge, since divergence is represented by the empty set.

$$\begin{array}{rrcl} \mathit{Prog}: & c & ::= & x := t \mid c; c \mid c + c \mid \textbf{if } b \textbf{ then } c \textbf{ else } c \mid \textbf{while } b \textbf{ do } c \\ \mathit{HT}: & h & ::= & \{\varphi\} C \{\psi\} \\ \Phi: & a & ::= & \varphi \mid h \end{array}$$

Figure 8.1: $\mathcal{L}_{HL}$, the language of Hoare Logic.

8.2 Consequence Relations

Like for First Order Logics, we can define (at least) two consequence relations for Hoare Logic, just by tailoring the general Definition 3.4.

Definition 8.1 *Let $\mathcal{M}$ range over f.o. structures and $[\![\cdot]\!]_{\mathcal{M}}$ the interpretation of Hoare Logic, as above. We define the* truth consequence relations *as follows:*

- *the* truth CR with respect to $\mathcal{M}$ *is*

$$\Gamma \models_{\mathcal{M}} a \iff (\forall \rho. [\![\Gamma]\!]_{\mathcal{M}} \rho = \top \Rightarrow [\![a]\!]_{\mathcal{M}} \rho = \top)$$

- *let Λ be a set of first-order structures for HL; the* truth CR with respect to Λ *is*

$$\Gamma \models_{\Lambda} a \iff (\forall \mathcal{M} \in \Lambda \forall \rho. [\![\Gamma]\!]_{\mathcal{M}} \rho = \top \Rightarrow [\![a]\!]_{\mathcal{M}} \rho = \top)$$

- *the* truth CR *is $\models \stackrel{\text{def}}{=} \bigcap_{\mathcal{M}} \models_{\mathcal{M}}$, where $\mathcal{M}$ ranges over all the first-order structures for HL; in other words,*

$$\Gamma \models a \iff \forall \mathcal{M} \forall \rho. [\![\Gamma]\!]_{\mathcal{M}} \rho = \top \Rightarrow [\![a]\!]_{\mathcal{M}} \rho = \top$$

We define the validity consequence relations *as follows:*

- *the* validity CR with respect to $\mathcal{M}$ *is*

$$\Gamma \models\!\!\models_{\mathcal{M}} a \iff (\forall \rho. [\![\Gamma]\!]_{\mathcal{M}} \rho = \top) \Rightarrow (\forall \rho. [\![a]\!]_{\mathcal{M}} \rho = \top)$$

- *let Λ be a set of first-order structures for HL; the* validity CR w.r.t. Λ *is*

$$\Gamma \models\!\!\models_{\Lambda} a \iff \forall \mathcal{M} \in \Lambda. (\forall \rho. [\![\Gamma]\!]_{\mathcal{M}} \rho = \top) \Rightarrow (\forall \rho. [\![a]\!]_{\mathcal{M}} \rho = \top)$$

- *the* validity CR *is $\models\!\!\models \stackrel{\text{def}}{=} \bigcap_{\mathcal{M}} \models\!\!\models_{\mathcal{M}}$, where $\mathcal{M}$ ranges over all the first-order structures for HL; in other words,*

$$\Gamma \models\!\!\models a \iff \forall \mathcal{M} (\forall \rho. [\![\Gamma]\!]_{\mathcal{M}} \rho = \top) \Rightarrow (\forall \rho. [\![a]\!]_{\mathcal{M}} \rho = \top)$$

Our terminology is slightly different from the standard one. In fact, the *truth* interpretation defined in [Apt81, LS88, AO91, Cou90] corresponds to our *validity* interpretation. In [Apt81, LS88, AO91, Cou90], a triple $\{\varphi\} c \{\psi\}$ is said to be true (under a model $\mathcal{M}$) if for all assignments $s, s' \in \mathit{Env}_{\mathcal{M}}$, if $s \models_{\mathcal{M}} \varphi$ and $[\![c]\!] s = s'$, then $s' \models \psi$. Furthermore, in

[LS88] a triple h is called a "logical consequence" of a set Γ of formulæ if $[\![h]\!]_{\mathcal{M}} = Env_{\mathcal{M}}$ for all models $\mathcal{M}$ of Γ. That is, given a model $\mathcal{M}$, if every assumption of Γ is true for all environments, then also h is true for all environments. This is exactly what we call the *validity* consequence relation, $\Vdash$.

A natural question arises: which is the most expressive CR for Hoare Logic? Consider Γ, h such that $\Gamma \Vdash h$. This asserts that if all formulæ in Γ are true *for all* environments, then for all environment h is true. If one of the assumptions is not true for all environments, then we can take as h any triple and still get a correct consequence. Therefore, if we want to have non-trivial consequences, Γ has to be true for *all* environments. This has several consequences.

- the only meaningful conclusions are *theorems*, that is the assertions a s.t. $\emptyset \Vdash a$. In fact, since $\Vdash$ is transitive (the cut rule is admissible), if $\Gamma \Vdash a$ and $\Vdash \Gamma$ then $\Vdash a$.
- a sort of "deduction theorem" for HL fails, that is if $\Gamma, \varphi \Vdash \{\varphi'\}c\{\psi\}$ we cannot say that $\Gamma \Vdash \{\varphi \wedge \varphi'\}c\{\psi\}$.
- in the LF internalization [AHMP92, Section 6.1], we can prove the goal

  $$\vdash_o (x = 0) \rightarrow \vdash_h (\{x + 1 = 1\}x := x + 1\{x = 0\})$$

  whose naïve meaning is "if x equals zero, then after adding 1 to x still x equals zero", which is clearly strange.[1]

On the other hand, the truth CR does not share these problems. If $\Gamma \models a$, then for every environment, if Γ holds then a holds. The hypotheses can be viewed as expressing conditions on the environment in which we want to infer the conclusion. In particular, if $\Gamma \models \{\varphi\}c\{\psi\}$, all the assumptions of Γ play the same rôle of φ. The intuitive meaning is: for all environments, if Γ and φ are satisfied *before* the execution of c, then ψ is satisfied *after* the execution of c (if it terminates). The "deduction theorem" holds for $\models$:

Proposition 8.1 *For φ, φ', ψ f.o. formulæ, Γ set of f.o. formulæand c command:*

$$\Gamma, \varphi \models \{\varphi'\}c\{\psi\} \iff \Gamma \models \{\varphi \wedge \varphi'\}c\{\psi\}$$

Therefore, like preconditions, assumptions can be viewed as constraints on environments; unlike preconditions, however, assumptions give environmental informations which are true in a whole (sub)derivation, not only in a single assertion. Hence, assumptions are more flexible than preconditions. This can be exploited in ND-style calculi where environmental information can be put in form of assumptions. This is in fact a general principle in ND calculi, and it applies to every kind of contextual information. For instance, Figure 8.2 shows three rules for dealing with procedures declaration and execution with both call-by-value and call-by-value/result argument passing [Plo81]. These rules are sound only with respect to the truth CR, and not with respect to the validity.

From a theoremhood point of view, the two classes of CR of Definition 8.1 are identical, i.e. $\forall \mathcal{M} : \models_{\mathcal{M}} a \iff \Vdash_{\mathcal{M}} a$.

[1] Actually, in [AHMP92, Section 6.2] an alternative signature for HL with a finite set of location is presented, which does not suffer of this drawback.

$$\textsc{ProcDecl}\ \frac{\begin{array}{c}(P'(x)=c)\\ \vdots\\ \{\varphi\}d[P'/P]\{\psi\}\end{array}}{\{\varphi\}\textbf{procedure}\ P(x)=c\ \textbf{in}\ d\{\psi\}}\ P' \text{ is a new procedure identifier}$$

$$\textsc{Call\_By\_Value}\ \frac{\begin{array}{c}(x'=t)\\ \vdots\\ P(x)=c\ \ \{\varphi\}c[x'/x]\{\psi\}\end{array}}{\{\varphi\}P(t)\{\psi\}}\ x' \text{ is a new identifier}$$

$$\textsc{Call\_By\_ValResult}\ \frac{\begin{array}{c}(x'=t)\\ \vdots\\ P(x)=c\ \ \{\varphi\}c[x'/x];x:=x'\{\psi\}\end{array}}{\{\varphi\}P(t)\{\psi\}}\ x' \text{ is a new identifier}$$

$D[P'/P]$ ($c[x'/x]$) denotes the metalinguistic substitution of P with P' (x with x')

Figure 8.2: Some ND-style rules for procedures in Hoare Logic.

8.3 Proof Systems

In the literature there is a well-known Hilbert-style proof system for Hoare Logic, [Apt81, AO91, LS88]; let $\vdash_{\mathbf{HS}HL}$ be the CR defined by this system. Similarly to the case of finitary systems for Dynamic Logic (Section 7.3.1), it is well-known that completeness in the sense of Definition 3.3 cannot be achieved; the completeness notion adopted in this case is the *arithmetical* (or *relative*) completeness (called also *Cook's Completeness*): completeness with respect to arithmetical models is achieved by extending the system with an oracle for the theory of the set of assumptions (see [Apt81, AO91, Har84], and Section 7.3.1). More formally, let A (capital α) be the class of arithmetical models, and $\mathrm{Th}(\Gamma,\models) \stackrel{\text{def}}{=} \{\varphi \in \mathit{Form} \mid \Gamma \models \varphi\}$, where $\models$ is a generic CR. Then

Proposition 8.2 *For all* $\Gamma \subset \Phi, a \in \Phi$*:*

1. $\Gamma \vdash_{\mathbf{HS}HL} a \Longrightarrow \Gamma \Vdash a$;
2. $\forall \mathcal{M} \in A : \Gamma \Vdash_{\mathcal{M}} a \Longrightarrow \Gamma \vdash_{\mathbf{HS}HL+\mathrm{Th}(\Gamma,\models_{\mathcal{M}})} a$.

Proof. An immediate extension of the usual proof of [Apt81]. □

The naïve attempt to define a ND-style system for $\models$ just by adding to the classical N.D. calculus for the FOL $\mathbf{N}PA^{\mathrm{qf}}$ the rules for Hoare Logic fails. In fact, the usual rules for composition, consequence and **while** are not sound with respect to $\models$. For instance, the following naïve composition rule

$$\frac{\Gamma \vdash \{\varphi\}c_1\{\theta\} \quad \Gamma \vdash \{\theta\}c_2\{\psi\}}{\Gamma \vdash \{\varphi\}c_1;c_2\{\psi\}}$$

allows us for derivations which are not sound w.r.t. the truth consequence relations, such as the following sequent:

$$\frac{\{x+1=1\}x := x+1\{true\} \quad \{true\}x := x\{x=0\}}{\{x+1=1\}x := x+1; x := x\{x=0\}}$$

every s such that $s(x) = 0$ satisfies the premises but not the conclusion. The problem is that the execution of the first command produces an environment which does not preserve the truth of the second assumption.

What we need is a condition ensuring that the execution of a program preserves the satisfaction of "following" assumptions. This is achieved by adopting a relation similar to Reynolds' non-interference judgement [Rey78]:

Definition 8.2 (Non Interference Judgements) *A relation* $\sharp \subseteq Prog \times \mathcal{P}(\Phi)$ *is a* non-interference judgement *if*

1. *it is syntactically detectable (it is decidable);*
2. $\forall c \in Prog, \forall \Gamma \in \mathcal{P}(\Phi) : c\sharp\Gamma \Rightarrow [\![c]\!]([\![\Gamma]\!]) \subseteq [\![\Gamma]\!]$;
3. $\forall c \in Prog : c\sharp\emptyset$.

It is important to notice that this kind of relations has a fail-safe character: there may be a $c \in Prog$ which does not interfere with a $\Gamma \subseteq \Phi$, but $c\sharp\Gamma$ does not hold. There are many possible non-interference judgement; actually, "there is a probably endless sequence of satisfactory definitions which come ever closer to the semantic relation of non-interference at the expense of increasing complexity" [Rey78]. Here we list some of these:

$$\begin{array}{lll} c\sharp_0\Gamma & \iff & \mathit{false} \\ c\sharp_1\Gamma & \iff & \Gamma = \emptyset \\ c\sharp_2\Gamma & \iff & \mathrm{V}(c) \cap \mathrm{V}(\Gamma) = \emptyset \end{array} \qquad \begin{array}{lll} c\sharp_3\Gamma & \iff & \mathrm{FV}(c) \cap \mathrm{FV}(\Gamma) = \emptyset \\ c\sharp_4\Gamma & \iff & \mathbb{AV}(c) \cap \mathrm{FV}(\Gamma) = \emptyset \\ c\sharp_\omega\Gamma & \iff & [\![c]\!]([\![\Gamma]\!]) \subseteq [\![\Gamma]\!] \end{array}$$

where V is the set of all variables, FV is the set of free variables, $\mathbb{AV}$ is the set of assignable variables (those appearing on the l.h.s. of assignments). These judgements are listed in increasing order of expressive power, since $\forall i \leq j < 3 : \sharp_i \subseteq \sharp_j$. (Actually, the first and the last ones are not proper non-interference judgements, since $c\sharp_0\emptyset$ does not hold, and $\sharp_\omega$ is not decidable. Indeed, $\sharp_\omega$ is exactly what we want to approximate; see Definition 8.2).

Let $\mathbf{N}HL$ be the ND-style system obtained by extending $\mathbf{N}PA$ with the rules Ass, AssComp', If, While', Or, Cons' (Figure 8.3). $\mathbf{N}HL$ is sound and relative complete with respect to $\models$

Proposition 8.3 *For all* $\Gamma \subset \Phi, a \in \Phi$*:*

1. $\Gamma \vdash_{\mathbf{N}HL} a \Longrightarrow \Gamma \models a$;
2. $\forall \mathcal{M} \in E : \Gamma \models_{\mathcal{M}} a \Longrightarrow \Gamma \vdash_{\mathbf{N}HL + \mathrm{Th}(\Gamma, \models_{\mathcal{M}})} a$.

Proof. (1. Soundness) it is easy to prove that Comp', While', Cons' are sound, by virtue of the side-conditions. We examine only the Comp' case. Let $\mathcal{M}$ be a model for HL, and suppose that

$H_1 : \Gamma_1 \models_{\mathcal{M}} \{\varphi\}c_1\{\theta\}$, that is $[\![\Gamma_1]\!]_{\mathcal{M}} \subseteq [\![\{\varphi\}c_1\{\theta\}]\!]_{\mathcal{M}}$;
$H_2 : \Gamma_2 \models_{\mathcal{M}} \{\theta\}c_2\{\psi\}$, that is $[\![\Gamma_2]\!]_{\mathcal{M}} \subseteq [\![\{\theta\}c_2\{\psi\}]\!]_{\mathcal{M}}$;
$H_3 : c_1\sharp\Gamma_2$.

$$\textsc{Ass}\ \frac{}{\Gamma \vdash \{\varphi[t/x]\}x := t\{\varphi\}}$$

$$\textsc{If}\ \frac{\Gamma \vdash \{\varphi \wedge b\}c_1\{\psi\} \quad \Gamma \vdash \{\varphi \wedge \neg b\}c_2\{\psi\}}{\Gamma\{\varphi\}\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2\{\psi\}}$$

$$\textsc{Or}\ \frac{\{\varphi\}c_1\{\psi\} \quad \{\varphi\}c_2\{\psi\}}{\{\varphi\}c_1 + c_2\{\psi\}}$$

$$\textsc{Cons'}\ \frac{\Gamma_1 \vdash \varphi \supset \varphi_1 \quad \Gamma_2 \vdash \{\varphi_1\}c\{\psi_1\} \quad \Gamma_3 \vdash \psi_1 \supset \psi}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \{\varphi\}c\{\psi\}}\ c\sharp\Gamma_3$$

$$\textsc{While'}\ \frac{\Gamma \vdash \{\varphi \wedge b\}c\{\varphi\}}{\Gamma \vdash \{\varphi\}\textbf{while } b \textbf{ do } c\{\varphi \wedge \neg b\}}\ c\sharp\Gamma$$

$$\textsc{Comp'}\ \frac{\Gamma_1 \vdash \{\varphi\}c_1\{\theta\} \quad \Gamma_2 \vdash \{\theta\}c_2\{\psi\}}{\Gamma_1, \Gamma_2 \vdash \{\varphi\}c_1; c_2\{\psi\}}\ c_1\sharp\Gamma_2$$

Figure 8.3: The rules of the system **N***HL*.

We have to prove that $\Gamma_1, \Gamma_2 \models_{\mathcal{M}} \{\varphi\}c_1; c_2\{\psi\}$. Let s be an environment such that $H_4 : s \in [\![\Gamma_1, \Gamma_2]\!]_{\mathcal{M}}$ and $H_5 : s \in [\![\varphi]\!]_{\mathcal{M}}$; now, the goal is $[\![c_1; c_2]\!]_{\mathcal{M}}s \subseteq [\![\psi]\!]_{\mathcal{M}}$. From H_1 and H_4 follows that $s \in [\![\{\varphi\}C_1\{\theta\}]\!]_{\mathcal{M}}$. From H_5, then, it is that $[\![c_1]\!]_{\mathcal{M}}s \subseteq [\![\theta]\!]_{\mathcal{M}}$.

Now, from H_4 we know that $s \in [\![\Gamma_2]\!]_{\mathcal{M}}$; since c_1 does not interfere with Γ_2 (for H_3, the side condition), we have that $[\![c_1]\!]_{\mathcal{M}}s \subseteq [\![\Gamma_2]\!]_{\mathcal{M}}$.

Hence, for H_2 we have $[\![c_2]\!]_{\mathcal{M}}([\![c_1]\!]_{\mathcal{M}}s) \subseteq [\![\psi]\!]_{\mathcal{M}}$, that is, $[\![c_1; c_2]\!]_{\mathcal{M}}s \subseteq [\![\psi]\!]_{\mathcal{M}}$.

(2.) Relative completeness is proved by induction on the syntax of the command involved in the Hoare triple, adapting one of the proofs for **HS***HL* (see e.g. [Apt81]) to the ND-style calculus **N***HL*. □

8.4 From ND-Style *HL* to *DL*

As we are going to explain in this section, the truth Hoare Logic can be seen as the link between Hoare Logic and Dynamic Logic.

8.4.1 Proof rules induced by non-interference judgements

Actually, the system **N***HL* is parametric in $\sharp$, and the completeness result (Proposition 8.3) holds for every non-interference judgement we adopt. However, if we want to encode effectively this system we *do* have to choose a definition for $\sharp$. Among many choices, the definition $\sharp_1$ is "minimal" in the sense that it is exactly what is needed in the proof of soundness and completeness (Proposition 8.3). More complex definitions are "more expressive", that is they perform a finer check of non-interference and therefore they allow for more general derivations. (Of course, definitions even more expressive than $\sharp_4$ can be given, e.g. based on data-flow analysis.) However, the more expressive is the side condition, the more cumbersome is its formalization and implementation, specially in type-theory based logical frameworks such as CIC or LF. We explored the case of $\sharp_1$. Let **N***HL*′

$$\textsc{Comp}''\ \frac{\Gamma \vdash \{\varphi\}C_1\{\theta\} \quad \vdash \{\theta\}c_2\{\psi\}}{\Gamma \vdash \{\varphi\}c_1; c_2\{\psi\}}$$

$$\textsc{While}''\ \frac{\vdash \{\varphi \wedge b\}c\{\varphi\}}{\vdash \{\varphi\}\mathbf{while}\ b\ \mathbf{do}\ c\{\varphi \wedge \neg b\}}$$

$$\textsc{Cons}''\ \frac{\Gamma_1 \vdash \varphi \supset \varphi_1 \quad \Gamma_2 \vdash \{\varphi_1\}c\{\psi_1\} \quad \vdash \psi_1 \supset \psi}{\Gamma_1, \Gamma_2 \vdash \{\varphi\}c\{\psi\}}$$

Figure 8.4: The three special rules of $\mathbf{N}HL'$.

$$\supset\text{-E}\frac{\begin{array}{c}(\varphi)\\ \vdots\\ \psi \quad \{\varphi \supset \psi\}c\{\theta\}\end{array}}{\{true\}c\{\theta\}} \qquad \wedge\text{-I}\frac{\begin{array}{c}(\psi)\\ \vdots\\ \{\varphi\}c\{\theta\}\end{array}}{\{\varphi \wedge \psi\}c\{\theta\}}$$

$$\supset\text{-I}\frac{\varphi \quad \{\psi\}c\{\theta\}}{\{\varphi \supset \psi\}c\{\theta\}} \quad \wedge\text{-El}\frac{\{\varphi \wedge \psi\}c\{\theta\} \quad \varphi}{\{\psi\}c\{\theta\}} \quad \wedge\text{-Er}\frac{\{\varphi \wedge \psi\}c\{\theta\} \quad \psi}{\{\varphi\}c\{\theta\}}$$

$$\vee\text{-I}\frac{\{\varphi\}c\{\theta\} \quad \{\psi\}c\{\theta\}}{\{\varphi \vee \psi\}c\{\theta\}} \quad \vee\text{-El}\frac{\{\varphi \vee \psi\}c\{\theta\} \quad \varphi}{\{true\}c\{\theta\}} \quad \vee\text{-Er}\frac{\{\varphi \vee \psi\}c\{\theta\} \quad \psi}{\{true\}c\{\theta\}}$$

Figure 8.5: Some rules admitted by the system $\mathbf{N}HL$.

be the system obtained from $\mathbf{N}HL$ by choosing $\sharp = \sharp_1$ (Figure 8.4); the three impure rules take the form of "proof rules," similar to the box-introduction rule $\Box'$-I of Modal Logic (Figure 5.2). Therefore, in encoding $\mathbf{N}HL'$ we have to represent modal features which are common to Modal Logic, and more specifically to Dynamic Logic; the techniques we will use in this venture will be the same of those developed for Modal Logics and adopted in the case of Dynamic Logic. Actually, an encoding of $\mathbf{N}HL'$ is derivable from the encoding of Dynamic Logic (see also Proposition 8.4).

8.4.2 Preconditions vs. Assumptions.

The calculus for $\models$ suggests several admissible rules; see e.g. Figure 8.5. All these rules are easily proved to be sound with respect to $\models$. Notice that $\wedge$-I and $\supset$-E are not sound with respect to $\|\!=$; actually, $\wedge$-I is nothing but the "deduction theorem" above-mentioned.

In this way we can merge the ND calculus for FOL and the rules for Hoare triples. This points out again that there is no conceptual difference between preconditions and assumptions, as it is suggested by the admissible rule

$$\frac{\begin{array}{c}(\varphi)\\ \vdots\\ \{true\}c\{\psi\}\end{array}}{\{\varphi\}c\{\psi\}} \qquad \text{which resembles the } \supset\text{-intro rule} \qquad \frac{\begin{array}{c}(\varphi)\\ \vdots\\ \psi\end{array}}{\varphi \supset \psi}$$

The equivalence between preconditions and assumptions is also suggested by the semantic

equality $[\![\{\varphi\}c\{\psi\}]\!] = [\![\varphi \supset [c]\,\psi]\!]$, and by the fact that implications are both "equivalent" to assumptions, in a logic of truth (but they *aren't*, in a logic of validity). Hence, it is semantically natural to drop the notion of precondition altogether, and to adopt a one-place connective for commands; this takes us into the language of Dynamic Logic.

8.4.3 Derivation of (Truth) *HL* in *DL*

In fact, the system for the truth Hoare Logic is easily derived in $\mathbf{N}DL^{\mathrm{qf}}$, the quantifier-free fragment of $\mathbf{N}DL$.

Theorem 8.4 *The rules* Comp", Cons", Or, If, While" *are derivable in* $\mathbf{N}PDL$*; The rule* Ass *is derivable in* $\mathbf{N}DL^{\mathrm{qf}}$*; the rule*

$$\textsc{While\_Termin}\ \frac{\vdash \varphi(n+1) \supset b \quad [\varphi(n+1)]c[\varphi(n)] \quad \vdash \varphi(0) \supset \neg b}{[\varphi(n)]\textbf{\textit{while}}\ b\ \textbf{\textit{do}}\ c[\varphi(0)]}\ n \notin \mathrm{FV}(c)$$

is derivable in $\mathbf{N}^a DL^{\mathrm{qf}}$.

Proof. We examine only the case of While" (Figure 8.3). Recall that **while** b **do** $c \stackrel{\text{def}}{=} (b?;c)^*;\neg b?$, and suppose that $\pi_h :\vdash \varphi \wedge b \supset [c]\,\varphi$. Then, for all $n \in \mathcal{N}$, we have $\varphi \vdash [b?;c]^n\,\varphi$, since $\pi_0 \stackrel{\text{def}}{=} \varphi \vdash \varphi$ and π_{n+1} is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{(p)_2 \qquad \dfrac{\varphi\ (b)_1}{\varphi \wedge b} \qquad \begin{matrix}\emptyset\\ \mathcal{D}_h\end{matrix}}{\begin{matrix}\pi_n\\ [b?;c]^n\,\varphi\end{matrix}} \qquad \dfrac{\varphi \wedge b \quad \varphi \wedge b \supset [c]\,\varphi}{[c]\,\varphi}}{[c]\,[b?;c]^n\,\varphi}(2);\dagger}{[b?]\,[c]\,[b?;c]^n\,\varphi}(1)}{[b?;c]\,[b?;c]^n\,\varphi}$$

where † is an application of Sc where $\Gamma = \{\varphi\}$. Then, the following derivation is a proof of While" in $\mathbf{N}PDL$.

$$\dfrac{\dfrac{\dfrac{\dfrac{(\varphi)_2\ (\neg b)_3}{\varphi \wedge \neg b}}{[\neg b?]\,(\varphi \wedge \neg b)}(3) \qquad \dfrac{\left\{\left.\begin{matrix}(\varphi)_1\\ \pi_n\\ [b?;c]^n\,\varphi\end{matrix}\right| n \in \mathcal{N}\right\}}{[(b?;c)^*]\,\varphi}\dagger}{[(b?;c)^*]\,[\neg b?]\,(\varphi \wedge \neg b)}(2);\ddagger}{[(b?;c)^*;\neg b?]\,(\varphi \wedge \neg b)}}{\varphi \supset [(b?;c)^*;\neg b?]\,(\varphi \wedge \neg b)}(1)$$

where †, ‡ are sound applications of *-I and Sc respectively. □

Actually, all these derivations have been carried out formally in the Coq environment; see Section 16.3.

Chapter 9

Propositional μ-calculus

The expressive power of Kozen's propositional modal μ-calculus [Koz83], often referred to as μK, subsumes many modal and temporal logics. In fact, many process logics such as PDL, CTL, CTL^*, $ECTL$ are strictly less expressive than μK. Despite its expressive power, μK enjoys nice properties such as decidability and the finite model property. The long-standing open problem of axiomatizability of μK has been solved by Walukiewicz [Wal95a], who has proved the completeness of Kozen's original system given in [Koz83]. We refer the interested reader to [Koz83, Wal95a, Wal95b].

Beside its expressive power and importance in the theory and verification of processes, the μ-calculus is interesting also for its syntactic and proof theoretic peculiarities, which we have not faced in the previous logics. These idiosyncrasies are mainly due to the negative formulæ constructor "μ", which resembles the "λ" of λ-calculus, and to context-sensitive production rules.

9.1 Syntax and Semantics

The language of μK is an extension of the syntax of Propositional Dynamic Logic (see Section 6.1). Let Act be a set of *actions* (ranged over by a, b, c), Φ_0 a set of atomic propositional letters (ranged over p, q), and Var a set of propositional variables (ranged over by x, y, z). Then, the syntax of the μ-calculus on Act is as follows:

$$\Phi: \quad \varphi \quad ::= \quad ff \mid p \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \supset \psi \mid [a]\,\varphi \mid x \mid \mu x\varphi$$

where ff is a distinguished propositional letter and the formation of $\mu x.\varphi$ is subject to the *positivity condition:* every occurrence of x in φ has to appear inside an even number of negations (In the following we will spell out this condition more in detail). The variable x is *bound* in $\mu x\varphi$; the usual conventions about α-equivalence apply. We write $\nu x\varphi$ as a shorthand for $\neg\mu x(\neg\varphi[\neg x/x])$.

Similarly to PDL, the interpretation of μ-calculus comes from Modal Logic. A model for the μ-calculus is a transition system, that is, a pair $\mathcal{M} = \langle S, [\![\cdot]\!]_{\mathcal{M}}\rangle$ where S is a (generic) nonempty set of *(abstract) states*, ranged over by s, t, r, and $[\![\cdot]\!]_{\mathcal{M}}$ is the interpretation of atomic propositional and command symbols: for all p, a, we have $[\![p]\!]_{\mathcal{M}} \subset S$ and $[\![a]\!]_{\mathcal{M}} : S \to \mathcal{P}(S)$.

Differently from *PDL* and *DL*, formulæ of μ-calculus may have free propositional variables; therefore, we need to introduce *environments*, which are functions assigning sets of states to propositional variables: $Env \stackrel{\text{def}}{=} Var \to \mathcal{P}(S)$. Given a model $\mathcal{M} = \langle S, [\![\cdot]\!] \rangle$ and an environment ρ, the semantics of a formula is the set of states in which it holds, and it is defined by extending $[\![\cdot]\!]$ compositionally, as follows:

$$
\begin{array}{ll}
[\![p]\!]_{\mathcal{M}}\rho \stackrel{\text{def}}{=} [\![p]\!] & [\![\varphi \wedge \psi]\!]_{\mathcal{M}}\rho \stackrel{\text{def}}{=} [\![\varphi]\!]_{\mathcal{M}}\rho \cap [\![\psi]\!]_{\mathcal{M}}\rho \\
[\![ff]\!]_{\mathcal{M}}\rho \stackrel{\text{def}}{=} \emptyset & [\![\varphi \supset \varphi]\!]_{\mathcal{M}}\rho \stackrel{\text{def}}{=} (S \setminus [\![\varphi]\!]_{\mathcal{M}}\rho) \cup [\![\psi]\!]_{\mathcal{M}}\rho \\
[\![x]\!]_{\mathcal{M}}\rho \stackrel{\text{def}}{=} \rho(x) & [\![[a]\,\varphi]\!]_{\mathcal{M}}\rho \stackrel{\text{def}}{=} \{s \in S \mid \forall r \in [\![a]\!]s : r \in [\![\varphi]\!]_{\mathcal{M}}\rho\} \\
[\![\neg\varphi]\!]_{\mathcal{M}}\rho \stackrel{\text{def}}{=} S \setminus [\![\varphi]\!]_{\mathcal{M}}\rho & [\![\mu x\varphi]\!]_{\mathcal{M}}\rho \stackrel{\text{def}}{=} \bigcap\{T \subseteq S \mid [\![\varphi]\!]_{\mathcal{M}}\rho[x \mapsto T] \subseteq T\}
\end{array}
$$

It is customary to view a formula φ with a free variable x as defining a function φ^ρ_x, as follows:

$$
\begin{array}{rccl}
\varphi^\rho_x(S): & \mathcal{P}(S) & \to & \mathcal{P}(S) \\
& U & \mapsto & [\![\varphi]\!]_{\mathcal{M}}\rho[x \mapsto U]
\end{array}
$$

The intuitive interpretation of $\mu x\varphi$ is then the *least fixed point* of φ^ρ_x. The condition on the formation of $\mu x\varphi$ ensures the existence of the lfp:

Proposition 9.1 *Let φ a formula and x a variable occurring only positively in φ. Then, in every environment ρ, φ^ρ_x has both the least and the greatest fixed point. In particular, the lfp of φ^ρ_x is $[\![\mu x\varphi]\!]\rho$.*

Proof. (Sketch) It is easy to show, by induction on the syntax of φ, that φ^ρ_x is monotone; the result thence follows from Knaster-Tarski's theorem. □

Notice that this result does not hold if we drop the condition on the formation of $\mu x\varphi$: for instance, the formula $\neg x$ identifies the function $(\neg x)^\rho_x(T) = S \setminus T$, which is not monotone and has no least fixed point.

9.2 Consequence Relations

The consequence relations we introduce here are extensions of the truth and validity CR's of propositional dynamic logic (Definition 6.1).

Definition 9.1 (Consequence Relations for μK) *Let $\mathcal{M}$ be a model for μK and $[\![\cdot]\!]_{\mathcal{M}}$ be the relative interpretation in $\mathcal{M}$. The* truth *and the* validity CR's *for μK wrt $\mathcal{M}$ are two relations $\models_{\mathcal{M}}, \Vdash_{\mathcal{M}} \subseteq \mathcal{P}(\Phi) \times \Phi$, defined as follows:*

$$
\begin{array}{rcl}
\Gamma \models_{\mathcal{M}} \varphi & \iff & \forall\rho.[\![\Gamma]\!]_{\mathcal{M}}\rho \subseteq [\![\varphi]\!]_{\mathcal{M}}\rho \\
\Gamma \Vdash_{\mathcal{M}} \varphi & \iff & \forall\rho.[\![\Gamma]\!]_{\mathcal{M}}\rho = S \Rightarrow [\![\varphi]\!]_{\mathcal{M}}\rho = S
\end{array}
$$

The (absolute) truth *and* validity CR's *for μK are defined as follows:*

$$
\begin{array}{rcl}
\Gamma \models \varphi & \iff & \forall\mathcal{M}.\Gamma \models_{\mathcal{M}} \varphi \\
\Gamma \Vdash \varphi & \iff & \forall\mathcal{M}.\Gamma \Vdash_{\mathcal{M}} \varphi
\end{array}
$$

ND-style	*sequent style*
μ-I $\dfrac{\varphi[(\mu x.\varphi)/x]}{\mu x.\varphi}$	$\dfrac{\Gamma \vdash \varphi[(\mu x.\varphi)/x]}{\Gamma \vdash \mu x.\varphi}$
μ-E $\dfrac{\mu x.\varphi \qquad \begin{array}{c}[\varphi[\psi/x]]\\ \vdots \\ \psi\end{array}}{\psi}$	$\dfrac{\Gamma \vdash \mu x.\varphi \quad \varphi[\psi/x] \vdash \psi}{\Gamma \vdash \psi}$

Figure 9.1: The specific rules of the system $\mathbf{N}\mu K$.

The finitary *truth consequence relations with respect to a model $\mathcal{M}$ is the restriction of $\models_{\mathcal{M}}$ to finite sets:*

$$\begin{array}{rcl} \Gamma \models_{f\,\mathcal{M}} \varphi & \iff & \exists \Delta \subseteq \Gamma,\ \textit{finite}.\Delta \models_{\mathcal{M}} \varphi \\ \Gamma \models_{f} \varphi & \iff & \exists \Delta \subseteq \Gamma,\ \textit{finite}.\Delta \models \varphi \end{array}$$

In the following, we will focus on the finitary truth consequence relation. For sake of simplicity, we will drop the $_f$, denoting by $\models$ the *finitary* truth CR $\models_f$.

9.3 Proof Systems

9.3.1 A ND-style system for μK

Usually, systems for μ-calculus are given in Hilbert style [Koz83, Sti92, And93]. Here we present a Natural Deduction style system for μK, namely $\mathbf{N}\mu K$. This system is composed by

- the system for classical propositional logic, $\mathbf{N}C$ (Figure 5.1);
- the rules for $\mathbf{N}K$ (Figure 5.2)
- the new two rules (introduction and elimination) for the new constructor μ, as presented in Figure 9.1.

These rules have a direct semantic interpretation.

- the introduction rule states that (the meaning of) $\mu x\varphi$ is a prefixed point of φ^{ρ}_{x};
- the elimination rule states that (the meaning of) $\mu x\varphi$ implies, and then "is less that", any prefixed point of φ^{ρ}_{x}.

Therefore, these rules state that (the meaning of) $\mu x\varphi$ is the minimum prefixed point, i.e. the least fixed point, of φ^{ρ}_{x}. Notice that the μ-elimination rule is a "proof rule": in fact, it can be stated equivalently as follows:

$$\frac{\Gamma \vdash \mu x.\varphi \quad \emptyset \vdash \varphi[\psi/x] \supset \psi}{\Gamma \vdash \psi}$$

PosinP $\dfrac{p \in \Phi_0}{posin(x,p)}$	NeginP $\dfrac{p \in \Phi_0}{negin(x,p)}$
PosinY $\dfrac{y \in Var}{posin(x,y)}$	NeginY $\dfrac{y \neq x}{negin(x,y)}$
PosinImp $\dfrac{negin(x,\varphi) \quad posin(x,\psi)}{posin(x,\varphi \supset \psi)}$	NeginImp $\dfrac{posin(x,\varphi) \quad negin(x,\psi)}{negin(x,\varphi \supset \psi)}$
PosinNeg $\dfrac{negin(x,\varphi)}{posin(x,\neg\varphi)}$	NeginNeg $\dfrac{posin(x,\varphi)}{negin(x,\neg\varphi)}$
PosinAnd $\dfrac{posin(x,\varphi) \quad posin(x,\psi)}{posin(x,\varphi \wedge \psi)}$	NeginAnd $\dfrac{negin(x,\varphi) \quad negin(x,\psi)}{negin(x,\varphi \wedge \psi)}$
PosinBox $\dfrac{posin(x,\varphi)}{posin(x,[a]\,\varphi)}$	NeginBox $\dfrac{negin(x,\varphi)}{negin(x,[a]\,\varphi)}$
PosinMu $\dfrac{\text{for } z \neq x : posin(x,\varphi[z/y])}{posin(x,\mu x\varphi)}$	NeginMu $\dfrac{\text{for } z \neq x : negin(x,\varphi[z/y])}{negin(x,\mu x\varphi)}$

Figure 9.2: The positivity proof system.

Here, the left subderivation has to depend on no assumptions, like to the necessitation rule of modal logic (rule $\Box'$-I, Figure 5.2).

The resulting system is then sound and complete with respect to the (finitary) truth consequence relation:

Theorem 9.2 *For Γ finite set of formulæ, φ formula: $\Gamma \vdash \varphi \iff \Gamma \models \varphi$.*

Proof. (Sketch) Soundness is proved by showing that each rule is sound. Completeness can be proved as follows. Since Γ is finite, $\Gamma \models \varphi \iff \models \bigwedge \Gamma \supset \varphi$. By completeness of Kozen's axiomatization [Wal95a, Wal95b], there is an Hilbert-style derivation of $\bigwedge \Gamma \supset \varphi$. Therefore, it is sufficient to prove that Kozen's axioms and rules (e.g. those presented in [Sti92]) are derivable in $\mathbf{N}\mu K$. □

9.3.2 A proof system for the positivity condition

Since we aim to encode the μ-calculus in some logical framework, we need to enforce the context-sensitive condition on the formation of formulæ of the form $\mu x\varphi$. That is, we ought to specify in detail the condition of "occurring positively in a formula" for a variable. This notion can be represented by two new judgements on formulæ and variables, *posin* and *negin*, which are derived by means of the rules in Figure 9.2. Roughly, $posin(x,\varphi)$ holds iff all occurrences of x in φ are positively; dually, $negin(x,\varphi)$ holds iff all occurrences of x in φ are negative. Notice that if x does not occur in φ, then it occurs both positively and negatively.

Let us formalize better the meaning of these auxiliary judgements. The notions they capture are the following:

Definition 9.2 (Monotonicity and Antimonotonicity) *Let φ be a formula and x a variable. We say that*

- φ *is* monotone *on* x *(written* $Mon_x(\varphi)$*) iff*

$$\forall \mathcal{M} \forall \rho \forall U, V \subseteq S: \ U \subseteq V \Rightarrow \varphi^\rho_x(U) \subseteq \varphi^\rho_x(V);$$

- φ *is* antimonotone *on* x *(written* $AntiMon_x(\varphi)$*) iff*

$$\forall \mathcal{M} \forall \rho \forall U, V \subseteq S: \ U \subseteq V \Rightarrow \varphi^\rho_x(U) \supseteq \varphi^\rho_x(V).$$

These notions refer directly to the semantic structures in which formulæ take meaning. The following result proves that the syntactic condition of positivity (respectively, negativity) captures correctly the semantic condition of monotonicity (respectively, antimonotonicity).

Proposition 9.3 *Let* φ *be a formula and* x *a variable. Then,*

1. $\vdash posin(x, \varphi) \Rightarrow Mon_x(\varphi)$*;*
2. $\vdash negin(x, \varphi) \Rightarrow AntiMon_x(\varphi)$*.*

Proof. By expliciting the definitions of *Mon* and *AntiMon*, the thesis is equivalent to the following statement:

given $\mathcal{M}$ model, ρ environment in $\mathcal{M}$, the following holds:

$$\begin{array}{lcl} \vdash posin(x, \varphi) & \Rightarrow & \forall U \subset V. \varphi^\rho_x(U) \subseteq \varphi^\rho_x(V) \\ \vdash negin(x, \varphi) & \Rightarrow & \forall U \subset V. \varphi^\rho_x(U) \supseteq \varphi^\rho_x(V). \end{array}$$

We prove this by simultaneous induction on the syntax of φ (which is equivalent to an induction on the proofs of $\vdash posin(x, \varphi)$, $\vdash negin(x, \varphi)$).
Base case: If $\vdash posin(x, \varphi)$ has height=0, then there are three subcases:

- $\varphi = p \in \Phi_0$: then, for $U \subseteq V : \varphi^\rho_x(U) = [\![p]\!] = \varphi^\rho_x(V)$.
- $\varphi = y$ different from x: then, for $U \subseteq V : \varphi^\rho_x(U) = \rho(y) = \varphi^\rho_x(V)$.
- $\varphi = x$: then for $U \subseteq V : x^\rho_x(U) = U \subseteq V = x^\rho_x(V)$.

If $\vdash negin(x, \varphi)$ has heigth=0, we proceed as above, but without the last case.
Inductive step: We see only two significant cases, the others being similar.

Let $\varphi = \varphi_1 \supset \varphi_2$. If $\vdash posin(x, \varphi_1 \supset \varphi_2)$, then there are two derivations $\vdash negin(x, \varphi_1)$, $\vdash posin(x, \varphi_2)$. By IH, we have $AntiMon_x(\varphi_1)$ and $Mon_x(\varphi_2)$, hence if $U \subseteq V$ then

$$(\varphi_1 \supset \varphi_2)^\rho_x(U) = (S \setminus \varphi^\rho_{1x}(U)) \cup \varphi^\rho_{2x}(U) \subseteq (S \setminus \varphi^\rho_{1x}(V)) \cup \varphi^\rho_{2x}(V) = (\varphi_1 \supset \varphi_2)^\rho_x(U)$$

If $\vdash negin(x, \varphi_1 \supset \varphi_2)$, then there are two derivations $\vdash posin(x, \varphi_1)$, $\vdash negin(x, \varphi_2)$. By IH, we have $Mon_x(\varphi_1)$ and $AntiMon_x(\varphi_2)$, hence if $U \subseteq V$ then

$$(\varphi_1 \supset \varphi_2)^\rho_x(U) = (S \setminus \varphi^\rho_{1x}(U)) \cup \varphi^\rho_{2x}(U) \supseteq (S \setminus \varphi^\rho_{1x}(V)) \cup \varphi^\rho_{2x}(V) = (\varphi_1 \supset \varphi_2)^\rho_x(U)$$

Let us consider the most complex case. If $\vdash posin(x, \mu z \varphi)$, then w.l.o.g. we can suppose z different from x, and that there is a derivations $\vdash posin(x, \varphi)$. Since

$$(\varphi^{\rho[z \mapsto W]}_x)(U) = [\![\varphi]\!]\rho[z \mapsto W, x \mapsto U]$$

we have $(\mu z\varphi)^\rho_x(U) = \bigcap \mathcal{A}(U)$, where

$$\mathcal{A}(U) \stackrel{\text{def}}{=} \{W \mid (\varphi^{\rho[z\mapsto W]}_x)(U) \subseteq W\}$$

Let $U \subseteq V$, and $W \in \mathcal{A}(V)$; then, $(\varphi^{\rho[z\mapsto W]}_x)(V) \subseteq W$. By IH,

$$(\varphi^{\rho[z\mapsto W]}_x)(U) \subseteq (\varphi^{\rho[z\mapsto W]}_x)(V) \subseteq W$$

and therefore $W \in \mathcal{A}(U)$. Hence, $\mathcal{A}(V) \subseteq \mathcal{A}(U)$, therefore

$$(\mu z\varphi)^\rho_x(U) = \bigcap \mathcal{A}(U) \subseteq \bigcap \mathcal{A}(V) = (\mu z\varphi)^\rho_x(V)$$

hence the thesis.

If we prove $\vdash \mathit{negin}(x, \mu x\varphi)$, we proceed as before, but, by antimonotonicity, $\mathcal{A}(V) \supseteq \mathcal{A}(U)$, therefore

$$(\mu z\varphi)^\rho_x(U) = \bigcap \mathcal{A}(U) \supseteq \bigcap \mathcal{A}(V) = (\mu z\varphi)^\rho_x(V)$$

hence the thesis. □

Notice that the converse of Proposition 9.3 does not hold. Consider e.g. $\varphi \stackrel{\text{def}}{=} (x \supset x)$: clearly, $[\![\varphi]\!]^\rho_{\mathcal{M}} = S$ always, and hence $(x \supset x)^\rho_x$ is both monotone and antimonotone. However, x does not occur only positively nor only negatively in φ. Correspondingly, we cannot derive $\vdash \mathit{posin}(x, (x \supset x))$ nor $\vdash \mathit{negin}(x, (x \supset x))$: by inspection of the proof system, the only rule for deriving $\mathit{posin}(x, (x \supset x)$ is PosinImp; hence, we should derive $\mathit{negin}(x, x)$, which is not possible (rule NeginY). A similar argument applies for $\mathit{negin}(x, (x \supset x))$.

This result can be generalized in the following limitation property:

Proposition 9.4 *If $x \in \mathrm{FV}(\varphi)$ occurs both positively and negatively in φ then neither $\mathit{posin}(x, \psi)$ nor $\mathit{negin}(x, \psi)$ are derivable.*

Proof. (Sketch) By induction on the syntax of φ. □

Part III

The Theory of Formal Representations

Chapter 10

Logical Frameworks

Type Theories, such as the Edinburgh Logical Framework [HHP93, AHMP92] or the Calculus of Inductive Constructions [CH88, Wer94] were especially designed, or can be fruitfully used, as a general logic specification language, i.e. as a Logical Framework (LF). In an LF, we can represent faithfully and uniformly all the relevant concepts of the inference process in a logical system: syntactic categories, terms, assertions, axiom schemata, rule schemata, tactics, etc. via the "judgements-as-types proofs-as-λ-terms" paradigm. The key concept is that of *hypothetico-general* judgement [Mar85], which is rendered as a type of the dependent typed λ-calculus of the Logical Framework. It allows to represent directly the assertions which occur in consequence relations (see Chapter 3) and to enforce immediately reflexivity, transitivity and monotonicity. Moreover, the λ-calculus metalanguage of an LF supports *higher order* syntax. Similarly to Martin-Löf's theory of arities (Chapter 2), every binding constructor of the object language can be represented by the only λ-abstraction of the metalanguage of the framework. Variables of the object language are hence identified with the variables of the metalanguage. As a result, α-conversion of bound variables is taken care of uniformly by the metalanguage. Moreover, "standard" substitution schemata, that is those like the one of λ-calculus, or the one of first-order logic, can be immediately delegated to the metalanguage, without the need of reimplementing them case-by-case. In this case, *instatiation* of both axioms and rule schemata can be delegated to the metalanguage; therefore, standard schematicities of consequence relations are immediately recovered from the schematicity of the metalanguage.

Logical Frameworks allow also for *higher-order* judgements, accordingly to Martin-Löf's theory of judgements. Hence, we can treat on a par axioms and rules, theorems and derived rules.

Encodings in LF's often provide the "normative" formalization of logic under consideration. The specification methodology of LF's, in fact, forces the user to make precise all tacit, or informal, conventions, which always accompany any presentation of a logic.

Any interactive proof development environment for the type theoretic metalanguage of an LF (e.g. Coq[CCF+95], LEGO [LPT89]), can be readily turned into one for a specific logic. We need only to fix a suitable environment (the *signature*), i.e. a declaration of typed constants corresponding to the syntactic sorts and term constructors with their arities, and the judgements and rule schemata of the logic (Chapters 2, 3). Such an LF-generated editor allows the user to reason "under assumptions" and go about in developing a proof the way mathematicians normally reason: using hypotheses, formulating conjectures, storing and retrieving lemmata, often in top-down, goal-directed fashion.

Moreover, Logical Frameworks provide a common medium for integrating different systems. Hence LF-derived editors rival special purpose editors when efficiency can be increased by integrating independent logical systems. LF-generated editors are *natural*. A user of the original logic can transfer immediately to them his practical experience and "trade tricks." The LF-derived editors do not force upon the user the overhead of unfamiliar indirect encodings, as would editors, say derived from FOL editors, via an encoding.

However, the wide conceptual universe provided by LF allows, on various occasions, to device genuinely new presentations of the logics. This has been the case for some of the Logics presented in this thesis. In particular, we shall capitalize on the feature of LF's of treating simultaneously different judgements and of treating proofs as first-class objects.

Structure of this Chapter. In Section 10.1 we will recall the main features of *Pure Type Systems*. The standard Natural-Deduction style presentation of the typing system, and the main properties of PTS's, will be given in Section 10.1.1. In Section 10.1.2 we introduce an alternative, Gentzen' sequent-style typing system for PTS's; this system is appealing for its proof-theoretic properties.

In Section 10.2 we present briefly the main concepts of the ELF$^+$ logical framework, which has been precisely designed in order to investigate the general notions of representation (encoding) of logics and proof systems. This section can be skipped without compromising the readibility of the rest of the thesis.

Sections 10.3 is devoted to one of the most important and common logical frameworks, the *Calculus of Inductive Constructions*. This logical framework will be adopted in the following chapters, in encoding the logics we have introduced in Part II. In Section 10.4, we will briefly recall the Coq system, which is the most common implementation of CIC.

In Section 10.5 we formalize the notion of "good representation" for both the syntactic and the deductive part of a formal system, accordingly to the "judgement-as-types" paradigm. We will present both the "proof-irrelevant" (*adequate*) and the "proof-relevant" (*natural*) notions of adequate representation of a proof system. Differently from the standard approach, we will focus on the representation of (multiple) consequence relations, instead of judgements. We will define a notion of *adequate representation* both for the language and the consequence relations of a logic. In the next chapter, we will see that the structural properties of PTS's and metatheoretic properties on logics to be represented are strictly tied up.

10.1 Pure Type Systems

In this section, we recall briefly the properties of Pure Type Systems.

The basic language is that of *preterms*:

Definition 10.1 (Preterms) *Let $\mathcal{S}$ a set of constant symbols, ranged over by s, and* Var *a set of variable symbols, ranged over by x, y, z. The* set of pre-λ-terms $\Lambda_{\mathcal{S}}$*, ranged over by M, N, A, B, C, is defined by the following abstract syntax:*

$$\Lambda_{\mathcal{S}} \qquad M ::= \quad s \mid x \mid MN \mid \lambda x{:}A.M \mid \Pi x{:}A.B$$

where the variable x is bound in $\lambda x{:}A.M$ and $\Pi x{:}A.B$.

For M a preterm, the set of free variables $\mathrm{FV}(M)$ *is defined as usual. Terms differing only on the name of bound variables are identified (α-equivalence).*

Definition 10.2 (Precontexts and Typing Judgements) *The* precontexts *for $\Lambda_{\mathcal{S}}$ are all and only the following lists of pairs:*

- *the empty context $\emptyset$ is a precontext; its domain is $\mathrm{dom}(\emptyset) = \emptyset$.*
- *if Γ is a precontext, x is a variable such that $x \notin \mathrm{dom}(\Gamma)$ and A is a preterm, then $\langle \Gamma, x : A \rangle$ is a precontext; its domain is $\mathrm{dom}(\langle \Gamma, (x, A) \rangle) = \mathrm{dom}(\Gamma) \cup \{x\}$*

We will write "$x_1 : A_1, \ldots, x_n : A_n$" instead of "$\langle \ldots \langle \emptyset, (x_1, A_1) \rangle, \ldots, (x_n, A_n) \rangle$".

A typing judgement *is a triple $\langle \Gamma, M, A \rangle$, written $\Gamma \vdash M : A$, where Γ is a precontext and M, A are preterms.*

10.1.1 Natural Deduction-style presentation

Following the traditional approach, the typing system we give in Definition 10.3 is in linearized Natural Deduction style.

Definition 10.3 (PTS) *A* Pure Type System *with β-conversion is specified by a triple $\langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$, where $\mathcal{S}$ is a set, $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ and $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$. The PTS that is given by a triple $\langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ is denoted by $\lambda_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$ and it is the λ-calculus $\Lambda_{\mathcal{S}}$ and typing rules appearing in Figure 10.1.*

If π is a derivation in $\lambda_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$ of the judgement $\Gamma \vdash M{:}A$, we write $\pi : (\Gamma \vdash M{:}A)$.

A Pure Type System *with $\beta\eta$-conversion is a PTS, where the equality condition*[1] *of the* CONVER *rule is $A =_{\beta\eta} B$.*

Since Church' seminal work [Chu40], the theory of typed λ-calculi and PTS's has been thoroughly investigated. For the sake of completeness, here we recall some of the main properties. The interested reader can find an extensive treatment of the metatheoretic properties of PTS's in [Bar92, Ber90, Geu93]. Detailed accounts for specific PTS's, often with different structural properties, can be found also in [Chu40, CH88, dB80, HHP93, Gar92]. In the rest of the section, we will consider only PTS with β-conversion.

Theorem 10.1 (Basic Properties of PTS's) *In any PTS $\lambda_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$, the following hold:*

Free Variables: *If $\Gamma \vdash M : A$ then $\mathrm{FV}(M) \cup \mathrm{FV}(A) \subseteq \mathrm{dom}(\Gamma)$;*

Transitivity: *If $(\forall (x : A) \in \Delta . \Gamma \vdash x : A)$, and $\Delta \vdash M : B$, then $\Gamma \vdash M : B$;*

Substitution Lemma: *If $\Gamma, x : A, \Delta \vdash M : C$ and $\Gamma \vdash N : A$ then $\Gamma, \Delta[N/x] \vdash M[N/x] : C[N/x]$;*

Thinning Lemma: *If $\Gamma \vdash M : A$ and $\Gamma \subseteq \Delta$ then $\Delta \vdash M : A$;*

[1] in this case, however, some care has to be taken in the formulation of the rules for the equality judgement. We do not discuss these syntoms here; the reader can find more details in [HHP93, Gar92].

Charateristic Rules		
SORT	$\emptyset \vdash s_1 : s_2$	$(s_1, s_2) \in \mathcal{A}$
PROD	$\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash M : s_2}{\Gamma \vdash \Pi x{:}A.M : s_3}$	$(s_1, s_2, s_3) \in \mathcal{R}$
Structural Rules		
START	$\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$	$x \notin \Gamma$
WEAK	$\dfrac{\Gamma \vdash A : s \quad \Gamma \vdash M : C}{\Gamma, x : A \vdash M : C}$	$x \notin \Gamma$
Logical Rules		
APP	$\dfrac{\Gamma \vdash N : A \quad \Gamma \vdash M : \Pi x{:}A.B}{\Gamma \vdash (MN) : B[N/x]}$	
ABS	$\dfrac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash \Pi x{:}A.B : s}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B}$	
Conversion Rule		
CONVER	$\dfrac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B}$	$A =_\beta B$

Figure 10.1: $\lambda_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$, the Natural Deduction-style typing system for PTS's.

Inversion Lemma: [2] *For Γ a context, C a term:*

1. $\Gamma \vdash c : C \Rightarrow \exists s \in \mathcal{S}.(C =_\beta s \wedge (c, s) \in \mathcal{A})$
2. $\Gamma \vdash x : C \Rightarrow \exists s \in \mathcal{S}, \exists B, \exists \Gamma_1, \Gamma_2.(B =_\beta C \wedge \Gamma = \Gamma_1, x : B, \Gamma_2 \wedge \Gamma_1 \vdash B : s)$
3. $\Gamma \vdash (\Pi x{:}A.B) : C \Rightarrow \exists (s_1, s_2, s_3) \in \mathcal{R}.(\Gamma \vdash A : s_1 \wedge \Gamma, x : A \vdash B : s_2 \wedge C =_\beta s_3)$
4. $\Gamma \vdash (\lambda x{:}A.M) : C \Rightarrow \exists s \in \mathcal{S}, \exists B.(\Gamma \vdash \Pi x{:}A.B : s \wedge \Gamma, x : A \vdash M : B \wedge C =_\beta \Pi x{:}A.B)$
5. $\Gamma \vdash (MN) : C \Rightarrow \exists A, B.(\Gamma \vdash M : \Pi x{:}A.B \wedge \Gamma \vdash N : A \wedge C =_\beta B[N/x])$

Uniqueness of types: *If $\Gamma \vdash M : A$ and $\Gamma \vdash M : B$, then $A =_\beta B$.*

Subject Reduction: *If $\Gamma \vdash M : A$ and $M \to_\beta N$, then $\Gamma \vdash N : B$; if $\Gamma \vdash M : A$ and $\Gamma \to_\beta \Delta$, then $\Delta \vdash N : B$;*

Strenghtening: *If $\Gamma, x : A, \Delta \vdash M : B$ and $x \notin \mathrm{FV}(\Delta, M, B)$, then $\Gamma_1, \Gamma_2 \vdash M : B$;*

Permutation: *If $\Gamma, x{:}A, y{:}B, \Delta \vdash M : A$ and $x \notin \mathrm{FV}(B)$ then $\Gamma, y{:}B, x{:}A, \Delta \vdash M : A$;*

For the proofs of these properties, see [Bar92, Geu93].

Particularly interesting cases arise when we consider only two sorts, that is $\mathcal{S} = \{*, \Box\}$, indented respectively as the sort of *terms* and the sort of *types*, and the only axiom $\mathcal{A} = \{(* : \Box)\}$. For each combination $s_1, s_2 \in \mathcal{S}$, there is a different production rule

[2] Also known as *generation lemma* [Bar92], *stripping lemma* [Geu93], *weak generation* [Gar92],...

(s_1, s_2, s_2), which we denote simply by (s_1, s_2). Apart from the basic $(*, *)$ (which allows us to abstract terms over terms), each of these rules can be admitted or not, giving rise to eight different systems. These systems form the so-called "λ-cube"; their properties have been thoroughly investigated in recent years (we refer to [Bar92, Geu93] for an comprehensive account). Their most noteworthy property is the *strong normalization:*

Theorem 10.2 (Strong Normalization for the λ-cube) *For every system in the λ-cube, if $\Gamma \vdash M : A$ then M, A are strong normalizing.*

Since the metatheory of PTS's in this form is well-known, we do not dwell further in these details. Instead, in the next section we introduce an alternative, and probably less known, presentations of PTS's, which turns to be interesting in view of the proof-theoretic use of Logical Frameworks as editors for top-down proof search.

10.1.2 Gentzen-style Pure Type Systems

In Figure 10.2, we present an alternative form for the typing systems of PTS's. This presentation is close to Gentzen' systems of sequents [Gen69]; in fact, there is a strict relation between these typing rules and the rules of sequent calculus, as shown in the following table:

Typing System	Sequent Calculus
START	reflexivity
WEAK	monotonicity
SUBST	cut, instantiation
APP'	left-introduction of implication and universal quantifier
ABS	right-introduction of implication and universal quantifier

Two particular cases of APP' are illuminating: if z does not appear free in C, the rule becomes

$$\prod\text{-L} \quad \frac{\Gamma \vdash t : A \quad \Gamma, z : B[t/x] \vdash M : C \quad \Gamma \vdash \Pi x{:}A.B : s}{\Gamma, y : \Pi x{:}A.B \vdash M[(yN)/z] : C} \quad y \notin \Gamma, z \notin \mathrm{FV}(C)$$

If we take the "proof irrelevance" version of the rule, by erasing the inhabitating terms, and replace Π by the universal quantificator, we obtain exactly Gentzen's $\forall$-left introduction:

$$\forall\text{-L} \quad \frac{\Gamma, B[t/x] \vdash C}{\Gamma, \forall x{:}A.B \vdash C}$$

We can go a step further: if x does not appear free in B, the product reduces to "implication", hence the rule becomes

$$\rightarrow\text{-L} \quad \frac{\Gamma \vdash N : A \quad \Gamma, z : B \vdash M : C \quad \Gamma \vdash A \rightarrow B : s}{\Gamma, y : A \rightarrow B \vdash M[(yN)/z] : C} \quad y \notin \Gamma, z \notin \mathrm{FV}(C)$$

By erasing the inhabitating terms and sort checkings, we obtain exactly Gentzen's $\supset$-left introduction:

$$\supset\text{-L} \quad \frac{\Gamma \vdash A \quad \Gamma, B \vdash C}{\Gamma, A \supset B \vdash C}$$

Characteristic Rules		
SORT	$\emptyset \vdash s_1 : s_2$	$(s_1, s_2) \in \mathcal{A}$
PROD	$\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash M : s_2}{\Gamma \vdash \Pi x{:}A.M : s_3}$	$(s_1, s_2, s_3) \in \mathcal{R}$
Structural Rules		
START	$\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$	$x \notin \Gamma$
WEAK	$\dfrac{\Gamma \vdash A : s \quad \Gamma \vdash M : C}{\Gamma, x : A \vdash M : C}$	$x \notin \Gamma$
SUBST	$\dfrac{\Gamma \vdash M : A \quad \Gamma, x : A \vdash N : B}{\Gamma \vdash N[M/x] : B[M/x]}$	
Logical Rules		
APP'	$\dfrac{\Gamma \vdash N : A \quad \Gamma, z : B[N/x] \vdash M : C \quad \Gamma \vdash \Pi x{:}A.B : s}{\Gamma, y : \Pi x{:}A.B \vdash M[(yN)/z] : C[(yN)/z]}$	$y \notin \Gamma$
ABS	$\dfrac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash \Pi x{:}A.B : s}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B}$	
Conversion Rule		
CONVER	$\dfrac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B}$	$A =_\beta B$

Figure 10.2: $\lambda^G_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$, the (Gentzen) sequent-style typing system for PTS's.

This presentation is closer in spirit to the activity of top-down proof searching. As we will see, if we disallow the SUBST rule Gentzen' style systems will type only *canonical* terms, that is terms in head normal form. Let us denote by $\lambda^-_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$ the PTS $\lambda^G_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$ without the SUBST rule (see Figure 10.3). Alternatively, $\lambda^-_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$ can be seen as $\lambda_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$ where the rule APP has been replaced by APP'.

It is well-known that the absence of "cut" rules and the restriction to the sole head normal forms reduces drastically the search space. A similar approach has been adopted by Pym and Wallen for the Edinburgh LF in investigating the proof search strategies for that framework [PW90]. Moreover, since the activity of proof search is goal-driven, the elementary proof tactics of most proof assistants normally correspond to an "inversion" of the rules (e.g. the `Apply` and `Intro` of `Coq`).

Similarly to what happens in sequent calculus, the SUBST rule allow us to derive also terms which are not in canonical form. Indeeed, the two presentations $\lambda_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$ and $\lambda^G_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$ are equivalent, as we are going to prove.

We proceed as follows. Lemma 10.3 and Lemma 10.4 are two technical results needed in the following propositions. Proposition 10.5 proves that sequent style PTS's subsume those in ND-style, since the APP rule is admissible. On the other hand, ND-style PTS's subsume those in sequent style, since both the APP' and the SUBST rules are admissible in any $\lambda_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$ (Proposition 10.6). Therefore, the two versions are equivalent, as far as every term, also those not in normal form, are concerned (Theorem 10.7).

Characteristic Rules		
SORT	$\emptyset \vdash s_1 : s_2$	$(s_1, s_2) \in \mathcal{A}$
PROD	$\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash M : s_2}{\Gamma \vdash \Pi x{:}A.M : s_3}$	$(s_1, s_2, s_3) \in \mathcal{R}$
Structural Rules		
START	$\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$	$x \notin \Gamma$
WEAK	$\dfrac{\Gamma \vdash A : s \quad \Gamma \vdash M : C}{\Gamma, x : A \vdash M : C}$	$x \notin \Gamma$
Logical Rules		
APP'	$\dfrac{\Gamma \vdash N : A \quad \Gamma, z : B[N/x] \vdash M : C \quad \Gamma \vdash \Pi x{:}A.B : s}{\Gamma, y : \Pi x{:}A.B \vdash M[(yN)/z] : C[(yN)/z]}$	$y \notin \Gamma$
ABS	$\dfrac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash \Pi x{:}A.B : s}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B}$	
Conversion Rule		
CONVER	$\dfrac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B}$	$A =_\beta B$

Figure 10.3: $\lambda^-_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$, the cut-free sequent-style typing system for PTS's.

Lemma 10.3 *In $\lambda^G_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$, if $\pi : (\Gamma \vdash M : \Pi x{:}A.B)$ then there is C such that $C =_\beta \Pi x{:}A.B$, and $\pi' : (\Gamma \vdash C : s)$ for some $s \in \mathcal{S}$.*

Proof. By induction on the height of π.
Base Case: π cannot be an axiom, hence the base case is vacuously true.
Inductive Case: by cases on the last rule applied.

Rules ABS,START,CONVER: trivially, π' is one of the subderivation.

Rule APP': then, there are $y, z, w, M', N, A', B', C, D, \Gamma'$ such that $M = M'[(yN)/z]$, $A = A'[(yN)/z]$, $B = B'[(yN)/z]$, and there are three derivations π_1, π_2, π_3 as follows:

$$\pi = \text{APP'}\frac{\overset{\pi_1}{\Gamma' \vdash N : C} \quad \overset{\pi_2}{\Gamma', z : D[N/w] \vdash M' : \Pi x{:}A'.B'} \quad \overset{\pi_3}{\Gamma' \vdash \Pi w{:}C.D : s'}}{\Gamma', y : \Pi w{:}C.D \vdash M'[(yN)/w] : \Pi x{:}A'[(yN)/w].B'[(yN)/w]}$$

Then, by inductive hypothesis, there is $\pi'_2 : (\Gamma', z : D[N/w] \vdash \Pi x{:}A'.B' : s)$ for some $s \in \mathcal{S}$. Now, $s[N/w] = s$, hence π' is as follows:

$$\pi' = \text{APP'}\frac{\overset{\pi_1}{\Gamma' \vdash N : C} \quad \overset{\pi'_2}{\Gamma', z : D[N/w] \vdash \Pi x{:}A'.B' : s} \quad \overset{\pi_3}{\Gamma' \vdash \Pi w{:}C.D : s'}}{\Gamma', y : \Pi w{:}C.D \vdash \Pi x{:}A.B : s}$$

Rule SUBST: then, there are y, M', N, A', B', C such that $M = M'[N/z]$, $A = A'[N/z]$, $B = B'[N/z]$, and there are three derivations π_1, π_2, π_3 as follows:

$$\pi = \text{SUBST}\frac{\overset{\pi_1}{\Gamma \vdash N : C} \quad \overset{\pi_2}{\Gamma, y : C \vdash M' : \Pi x{:}A'.B'}}{\Gamma \vdash M'[N/y] : \Pi x{:}A'[N/y].B'[N/y]}$$

Then, by inductive hypothesis, there is $\pi_2' : (\Gamma, y : C \vdash M' : \Pi x{:}A'.B')$ for some $s \in \mathcal{S}$. Now, $s[N/y] = s$, hence π' is as follows:

$$\pi' = \textsc{subst}\frac{\pi_1 \quad \pi_2'}{\Gamma \vdash N : C \quad \Gamma, y : C \vdash \Pi x{:}A'.B' : s}{\Gamma \vdash \Pi x{:}A.B : s}$$

□

Lemma 10.4 *In $\lambda^G_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$, let $\pi : (\Gamma \vdash M : \Pi x{:}A.B : s)$. Then, there is $(s_1, s_2, s) \in \mathcal{R}$ such that*

1. *there are two derivations $\pi_1 : \Gamma \vdash A : s_1$ and $\Gamma, x : A \vdash B : s_2$;*
2. *if $\Gamma \vdash N : A$, then $\Gamma \vdash B[N/x] : s_2$.*

Proof. 1. Straightforward induction on π.
2. Apply SUBST to the proof of $\Gamma, x : A \vdash B : s_2$ obtained in the previous point. □

The Natural-Deduction style version is subsumed by the sequent-style one:

Proposition 10.5 *The rule* APP *is admissible in $\lambda^G_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$.*

Proof. Let $\pi_1 : (\Gamma \vdash M : \Pi x{:}A.B)$ and $\pi_2 : (\Gamma \vdash N : A)$. Then, the following is a proof of $\Gamma \vdash (MN) : B[N/x]$:

$$\frac{\pi_1 \qquad \textsc{app}'\dfrac{\pi_2 \quad \dfrac{\pi_3}{\Gamma \vdash B[N/x] : s'}}{\Gamma, z : B[N/x] \vdash z : B[N/x]} \quad \pi_4}{\Gamma \vdash M : \Pi x{:}A.B \qquad \Gamma, y : \Pi x{:}A.B \vdash (yN) : B[N/x] \qquad \Gamma \vdash \Pi x{:}A.B : s}{\Gamma \vdash (MN) : B[N/x]}$$

where the last rule applied is SUBST, and π_3 and π_4 are obtained by Lemmata 10.4, 10.3 respectively. The variable y does not appear free in $B[N/x]$, because $\mathrm{FV}(B[N/x]) = \mathrm{FV}(\Pi x{:}A.B) \cup \mathrm{FV}(N) \subseteq \mathrm{dom}(\Gamma)$ (the last inclusion holds for $\Gamma \vdash N : A$ and $\Gamma \vdash \Pi x{:}A.B : s$), and y is chosen so that $y \notin \Gamma$. □

On the other hand, $\lambda_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$ subsumes $\lambda^G_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$:

Proposition 10.6 *The rules* APP' *and* SUBST *are admissible in $\lambda_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$.*

Proof. We have to prove that for every derivation in $\lambda^G_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$ whose last rule applied is either SUBST or APP', th is a corresponding derivation in $\lambda_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$. The proof proceeds by induction on the height of derivations. The base steps are vacuously verified.

Inductive steps: if the last rule applied is SUBST, then it is an application of the Substitution Lemma (Proposition 10.1).

Let us consider the APP case. Let $\pi_1 : (\Gamma \vdash N : A)$, $\pi_2 : (\Gamma, z : B[N/x] \vdash M : C)$ and $\pi_3 : (\Gamma \vdash \Pi x{:}A.B : s)$ proofs in $\lambda_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$; we have to prove that there is a derivation of $\Gamma, y : \Pi x{:}A.B \vdash M[(yN)/z] : B[(yN)/z]$.

From π_2 and π_3, there is a proof $\pi_2' : (\Gamma, y : \Pi x{:}A.B, z : B[N/x] \vdash M : C)$: we simply follow π_2, and just before the START rules for z we introduce y by means of WEAKenings. Moreover, let π_4 be the following derivation:

$$\pi_4 \stackrel{\text{def}}{=} \text{APP}\dfrac{\text{START}\dfrac{\overset{\pi_3}{\Gamma \vdash \Pi x{:}A.B : s}}{\Gamma, y : \Pi x{:}A.B \vdash y : \Pi x{:}A.B} \quad \text{WEAK}\dfrac{\overset{\pi_1}{\Gamma \vdash N : A} \quad \overset{\pi_3}{\Gamma \vdash \Pi x{:}A.B : s}}{\Gamma, y : \Pi x{:}A.B \vdash N : A}}{\Gamma, y : \Pi x{:}A.B \vdash (yN) : B[N/x]}$$

Then, by applying the Substitution Lemma of PTS's (Proposition 10.1) to π_4 and to π_2', we obtain the derivation of $\Gamma, y : \Pi x{:}A.B \vdash M[(yN)/z] : B[(yN)/z]$. □

Theorem 10.7 (Equivalence between ND- and sequent-style systems) *For any PTS specification $\langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$, $\lambda_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$ is equivalent to $\lambda_\beta^G(\mathcal{S}, \mathcal{A}, \mathcal{R})$, in the sense that for each derivation $\pi : (\Gamma \vdash M : A)$ in $\lambda_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$, there is $\pi' : (\Gamma \vdash M : A)$ in $\lambda_\beta^G(\mathcal{S}, \mathcal{A}, \mathcal{R})$, and vice versa.*

Proof. Immediate from Propositions 10.5, 10.6. □

As we have point out before, particularly interesting is the SUBST-free versions of sequent-style PTS. We will prove now that the derivations which can be carried out in these systems regard all and only terms in head normal form.

Lemma 10.8 (Left Conversion) *In $\lambda_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$, $\lambda_\beta^G(\mathcal{S}, \mathcal{A}, \mathcal{R})$, $\lambda_\beta^-(\mathcal{S}, \mathcal{A}, \mathcal{R})$, the following rule is admissible.*

$$\text{CONV-L}\dfrac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash M : B}\ A =_\beta C$$

Proof. Let π_1, π_2 such that $\pi_1 : (\Gamma, x : A \vdash M : B)$ and $\pi_2 : (\Gamma \vdash C : s)$. We can trace back the derivation π_1 until we reach the instances of START which introduce $(x : A)$ in the context. These instances appear always as the last rule of a subderivation π_3 of π_1 as follows:

$$\pi_3 \left\{ \dfrac{\overset{\pi_4}{\Gamma \vdash A : s'}}{\Gamma, x : A \vdash x : A} \right.$$

Now, we can carry out the following derivation $\pi_3' : (\Gamma, x : C \vdash x : A)$:

$$\text{CONVER}\dfrac{\text{START}\dfrac{\overset{\pi_2}{\Gamma \vdash C : s}}{\Gamma, x : C \vdash x : C} \quad \overset{\pi_4}{\Gamma \vdash A : s'}}{\Gamma, x : C \vdash x : A}$$

Finally, we obtain the proof of $\Gamma, x : C \vdash M : B$ by replacing all the subderivations π_3 in π_1 by π_3'. □

The following theorem shows that the three versions of PTS are equivalent, as far as only terms in head normal form are concerned.

Theorem 10.9 *Let $\langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ a PTS, Γ a context, and M, A terms. Then, the following are equivalent:*

1. $\Gamma \vdash M : A$ *in* $\lambda^-_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$;
2. $\Gamma \vdash M : A$ *in* $\lambda^G_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$ *and* M *is in head normal form;*
3. $\Gamma \vdash M : A$ *in* $\lambda_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$ *and* M *is in head normal form.*

Proof. $1 \Rightarrow 2$**.** Easy induction; just notice that the head of applications is always a variable (see the APP' rule), and then the final term is in head normal form.

$2 \Rightarrow 3$**.** By Proposition 10.6.

$3 \Rightarrow 1$**.** Let $\pi : (\Gamma \vdash M : A)$ in $\lambda_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$, and let M be in head normal form. We have to prove that there is a derivation $\pi' : (\Gamma \vdash M : A)$ in $\lambda^-_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$. We proceed by induction on π.
Base case. If π is an instance of SORT, then $\pi' \stackrel{\text{def}}{=} \pi$.
Inductive step. By cases on the last rule applied. All but one cases are trivial; we discuss the only interesting one. Let APP be the last rule applied; the derivation appers as follows:

$$\text{APP}\frac{\begin{matrix}\pi_1 & \pi_2\\ \Gamma \vdash N : C & \Gamma \vdash M' : \Pi x{:}C.B\end{matrix}}{\Gamma \vdash (M'N) : B[N/x]}$$

We cannot replace directly this rule by appling Proposition 10.5, because this would introduce an instance of SUBST, which is not available in $\lambda^-_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$. However, since M is in hnf, also M' and N are in hnf; moreover, M' cannot be an abstraction, otherwise $M'N$ would be reducible. Therefore, M' is of the form $(yN_1 \dots N_k)$, $k \geq 0$, for some $N_1, \dots, N_k$ terms in hnf. The whole derivation π appears as follows:

$$\text{APP}\frac{\begin{matrix}\pi_1 & \pi_2\\ \Gamma \vdash N : C & \Gamma \vdash (yN_1 \dots N_k) : \Pi x{:}C.B\end{matrix}}{\Gamma \vdash (yN_1 \dots N_k N) : B[N/x]}$$

By the Inversion Lemma, there are k subderivations such that for $\delta_i : \Gamma_i \vdash N_i : C_i$, and a subderivation $\delta_0 : \Gamma_0 \vdash y : \Pi x_1{:}C_1 \dots \Pi x_k{:}C_k.B_0$, where $\Gamma_0, \dots, \Gamma_k$ are initial segments of Γ, and $\Pi x{:}C.B = B_0[N_1/x_1] \dots [N_k/x_k]$. (These derivations δ_i are subderivations of π_2.) By inductive hypothesis, each of these derivations can be turned into a corresponding derivation in $\lambda^-_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$; then, by applying $k+1$ times the APP' rule we obtain the thesis. □

Notice that this result does not imply any kind of normalization; in fact, Theorem 10.9 deals only with normal forms, and does not state anything about the other terms. In our "logical" setting, normalization corresponds to the elimination of the SUBST rule. Hence, it can be stated as follows:

Theorem 10.10 (Cut Elimination) *For $\langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ a PTS specification, the following are equivalent:*

- $\langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ *is strongly normalizing;*
- *if $\pi : (\Gamma \vdash M : A)$ in $\lambda^G_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$, then for all M', A' hnf such that $M \to_\beta M'$, $A \to_\beta A'$, there is π' in $\lambda^-_\beta(\mathcal{S}, \mathcal{A}, \mathcal{R})$ such that $\pi' : (\Gamma \vdash M' : A')$.*

10.2 The logical framework ELF$^+$

An important advance in the area of logical frameworks is represented by the *Edinburgh Logical Framework*, designed in the late '80s by Harper, Honsell and Plotkin [HHP93]. This framework has been influenced by de Bruijn's AUTOMATH project [dB80] and Martin-Löf type theory [NPS92].

We present briefly a similar framework, namely Gardner's ELF$^+$ [Gar92, Gar93], which has been precisely designed in order to investigate general notions of representation (encoding) of logics and proof systems. Here we present only the type structure and the intended meaning of the sorts of ELF$^+$, since the *judgement-as-types* paradigm and the related notions of encodings will be presented in detail in Section 10.5. Therefore, this section can be skipped without compromising the readibility of the rest of the thesis.

ELF$^+$, as ELF, is a *PTS with $\beta\eta$-equality and signatures* [HHP93, Ber90, Gar92]. A *signature* is a set of constants whose types specify the encoded logic. Although signatures can be roughly seen as "contexts whose variables cannot be discharged", their formation rules are stronger than those of contexts. Indeed, there is an universe (Kind) which can be inhabitated by constants but not by variables; in other words, we can declare in a signature that a constant c has type Kind, but we cannot assume that a variable x inhabits Kind. Further details are in [HHP93, Gar92]. Hence, a PTS with signature is specified by a quadruple $\langle \mathcal{S}, \mathcal{V}, \mathcal{A}, \mathcal{R} \rangle$ where $\mathcal{S}$, $\mathcal{A}$, $\mathcal{R}$ are as in the case of simple PTS's (Definition 10.3); the new set $\mathcal{V}$ is a subset of $\mathcal{S}$ and specifies the sorts which can be inhabitated by context variables (see [Gar92, Gar93]).

The type theory of ELF$^+$ is a variant of the Edinburgh Logical Framework as presented in [HHP93]. ELF$^+$ differs from ELF on the universe structure: in place of the unique universe Type of the ELF, ELF$^+$ has three universes called Sort, Extra and Judge.

Definition 10.4 (ELF$^+$) *The framework ELF$^+$ is the PTS with $\beta\eta$-conversion and signatures specified by $\langle \mathcal{S}, \mathcal{V}, \mathcal{A}, \mathcal{R} \rangle$ where*

$$\begin{aligned}
\mathcal{S} &= \{\mathsf{Sort}, \mathsf{Extra}, \mathsf{Judge}, \mathsf{Kind}\} \\
\mathcal{V} &= \{\mathsf{Sort}, \mathsf{Extra}, \mathsf{Judge}\} \\
\mathcal{A} &= \{(\mathsf{Sort} : \mathsf{Kind}), (\mathsf{Extra} : \mathsf{Kind}), (\mathsf{Judge} : \mathsf{Kind})\} \\
\mathcal{R} &= \{(\mathsf{Sort}, \mathsf{Kind}, \mathsf{Kind}), (\mathsf{Extra}, \mathsf{Kind}, \mathsf{Kind}), (\mathsf{Judge}, \mathsf{Judge}, \mathsf{Extra})\} \cup \\
&\quad \cup \{(s_1, s_2, \mathsf{Extra}) \mid s_1, s_2 \in \{\mathsf{Sort}, \mathsf{Extra}\}\}
\end{aligned}$$

Each basic sort has a precise meaning:

- inhabitants of Sort represent the *syntact sorts* of the object language. In encoding a language, for each syntactic sort S, we declare a constant $T_S : \mathsf{Sort}$; each expression of sort S is therefore represented by a term of type T_S.
- inhabitants of Judge represent the *basic judgements* of the object logic, and hence its consequence relations. Each judgement A of the logic will be represented by a specific constant $J_A : \theta$; θ is a type which ends with Judge and reflects the arity of A. For instance, the "derivability" judgement T on a first-order language S is represented by a constant $J_T : T_S \to \mathsf{Judge}$.

- The extra universe Extra is used as a "scratch area", for defining terms which have no immediate correspondence with the object logic. It is important to notice that Π-abstractions of sorts, extra types and judgements all inhabit Extra. This is because Π-abstraction is viewed as a part of the machinery of the metalanguage, rather than of the object language.

The introduction of an additional universe for Π-abstractions reduces drastically the class of terms which can represent expressions and proofs of the object logic. For instance, terms of type Sort → Sort (that is, "schematic terms", terms with holes), do not represent any object expression, because the kind of Sort → Sort is Extra.

10.3 The Calculus of Inductive Constructions

In this section we will present briefly the *Calculus of (Inductive) Constructions* (CIC for short). A complete and detailed exposition of this framework is out of the scopes of this thesis; we refer hence the interested reader to the extensive literature [CH88, CP90, CCF+95, Hue92, Hue94, Pau93, Tay88, Wer94].

CIC as a PTS

In its original formulation, Coquand and Huet's *Calculus of Constructions (CC)* [CH88] can be defined as the PTS λC of Barendregt's λ-cube, that is

$$\lambda C \stackrel{\text{def}}{=} \boxed{\begin{array}{l} \mathcal{S} = \{\star, \Box\} \\ \mathcal{A} = \{\star : \Box\} \\ \mathcal{R} = \mathcal{S}^2 \end{array}}$$

Actually, the type structure of CIC is much more complex than the one of λC. In place of sort $\star$, CIC have two basic sorts, namely Prop and Set, which play a rôle similar to Judge and Sort in ELF$^+$, respectively.

The sort Prop is supposed to be the type of *logical proposition*, *predicates*, or judgements. Accordingly to the *proposition-as-types, proofs-as-terms* paradigm, if A has type Prop then it represents a logical proposition; the fact that A is inhabited by a term M represents the fact that A holds. Each term M inhabiting A represents a *proof* of A.

On the other hand, the sort Set is supposed to be the type of datatypes, such as naturals, lists, trees, booleans, etc. These types differ from those inhabiting Prop for their constructive contents, as we will see later.

Like any other term, also Prop and Set have to be given a type. Due to well-known consistency problems, we cannot inhabit an universe by itself, since this would yield Girard's paradox: every type would be inhabited, hence every proposition would be provable [Bar92]. Therefore, in CIC there are infinitely many other sorts, namely $\mathsf{Type}(i)$ and $\mathsf{Typeset}(i)$ for any integer i. These form two *hierarchies of universes*: Prop inhabits $\mathsf{Type}(0)$ which inhabits $\mathsf{Type}(1)$ which inhabits $\mathsf{Type}(2)$ and so on; similarly for Set. Indeed, CIC can be presented as the PTS identified by the following triple:

$$\text{CIC} \stackrel{\text{def}}{=} \boxed{\begin{array}{lcl} \mathcal{S} & = & \{\mathsf{Prop}, \mathsf{Set}\} \cup \{\mathsf{Type}(i), \mathsf{Typeset}(i) \mid i \in \mathcal{N}\} \\ \mathcal{A} & = & \{\mathsf{Prop}{:}\mathsf{Type}(0), \mathsf{Set}{:}\mathsf{Typeset}(0)\} \cup \\ & & \cup\{(\mathsf{Type}(i){:}\mathsf{Type}(i+1)), (\mathsf{Typeset}(i){:}\mathsf{Typeset}(i+1) \mid i \in \mathcal{N}\} \\ \mathcal{R} & = & \{\mathsf{Prop}, \mathsf{Set}\}^2 \cup \{(t(i), t'(j)) \mid t, t' \in \{\mathsf{Type}, \mathsf{Typeset}\},\ i, j \in \mathcal{N},\ i \leq j\} \end{array}}$$

Remark 1. It is important to notice that the reasons for introducing more than one basic type in CIC are different from those of ELF$^+$. As we have already observed in Section 10.2, the main reason for introducing three basic sorts in ELF$^+$ is to give an accurate formalization of the judgement-as-types paradigm. On the other hand, the existence of two distinct hierarchies of types in CIC is motivated by a peculiar feature of the Coq implementation, namely the *extraction of programs from proofs*, that is, the computational interpretation of proof objects, accordingly to the Curry-Howard isomorphism and Christine Paulin-Mohring's *realizability interpretation*. However, a proof may have no computational content; therefore, the two hierarchies are introduced in order to keep apart the constructive universes from those non constructive. No program will be extracted from a term inhabiting a sort of the Prop hierarchy; instead, any object $A : \mathsf{Set}$ is interpreted as a *specification*, and any term $M : A$ is interpreted as a program which meets the specification A. Coq implements a mechanism for extracting a (function) program from this object, in an effective way. We will do not dwell further on this topic, since it is out of the scopes of this thesis; we refer the interested reader to [CCF$^+$95, Pau89, PW93].

Remark 2. In the Coq implementation, the user does not need to specify the indexes of the type hierarchy. One only write "Type" or "Typeset", without any index. The system itself generates consistently a new internal index for each instance of these types, and checks that soundness is preserved. Hence, from the point of view of a Coq user, it is Type : Type and Typeset : Typeset, although these occurrences denote internally different sorts.

Inductive Definitions

An important feature of CIC is the possibility of defining types *inductively*; for instance, it is possible defining a set (e.g. a type inhabiting Set) by declaring that is the *least set* closed under the application of a given family of constructors. Actually, any extension of the second-order PTS $\lambda 2$, such as Girard's F [GLT90], allows for inductive definitions by means of higher-order (impredicative) quantifications, but these representations are not always satisfactory (see [Pau93] for a discussion). A way for overcoming this problem has been introduced in the Calculus of Inductive Constructions by Thierry Coquand and Christine Paulin-Mohring [CP90, Pau93]. The idea is to *extend* the language of typed λ-terms by adding some special constants which represent the definition, introduction and elimination of inductive types. Therefore, the language of pre-λ-terms $\Lambda_{\mathcal{S}}$ of Definition 10.1 is extended by adding the following term constructors:

$$\Lambda_{\mathcal{S}} : \quad M \quad ::= \quad \ldots \mid \mathit{Ind}(x{:}A)\{A_1, \ldots, A_n\} \mid \mathit{Constr}(i, A) \mid \mathit{Elim}(M, A)\{M_1, \ldots, M_n\}$$

where i ranges over integers. The intuitive meaning of these new terms is the following:

- $\mathit{Ind}(x{:}A)\{A_1, \ldots, A_n\}$ is an inductive type of sort A, denoted by x; $A_1, \ldots, A_n$ are the constructors of x;
- if $I = \mathit{Ind}(x{:}A)\{A_1, \ldots, A_n\}$, then $\mathit{Constr}(i, I)$ is a term inhabiting the inductive type I, and it is defined by the ith constructor of I;
- $\mathit{Elim}(c, P)\{M_1, \ldots, M_n\}$ is a term defined by an induction/recursion schema over an inductively defined type.

Of course, since we have extended the language, we have to extend the typing system accordingly. Beside the rules for typing the new terms, we need to introduce rules for stating the equivalence between these terms. Moreover, we need to represent the computational content of inductively defined datatypes: in fact, since we need to define functions by recursion, we need also a way for performing computation with these functions. Therefore, a new term reduction rule, called *ι-reduction*, is introduced beside the usual β-reduction. We do not insist further on this topic; we refer the interested reader to [CCF+95, Pau93, CP90].

10.4 The proof assistant Coq

One of the most common implementations of the Calculus of Inductive Constructions is the Coq proof assistant. A detailed presentation of Coq is outside the aims of this thesis; we refer the reader to the reference manual for a complete description and an extensive bibliography [CCF+95].

Coq is the result of about ten years of research of the Coq project, developed jointly by Gérard Huet's group at INRIA-Rocquencourt and Christine Paulin-Mohring's group at the ENS in Lyon. Roughly, it can be seen ad composed by three parts: the *logical (meta)language*, the *proof assistant* and the *program extractor*. We will not describe the latter one, since it is related to an issue which is not relevant to this thesis (the synthesis of computer programs from specifications declared in the logical language).

10.4.1 The metalogical language

The metalogical language of Coq, in which we describe the object logic, is the Calculus of Inductive Constructions (CIC), presented in Section 10.3. Of course, the reduced set of symbols available on textual terminals forces us to some changes to the usual notation of typed λ-calculi. The resulting language, called *Gallina*, is (partially) described by the BNF rules in Figure 10.4. By convention, phrases in Gallina are written in `verbatim`.

Some explanation about the meaning of each constructor is in order.

- The meaning of a phrase

  $$(term_0\ term_1\ \ldots\ term_n)$$

  is that of a function $term_0$ applied to the arguments $term_1, \ldots, term_n$.

- `(`*binder*`)`*term* means the λ-abstractions, and `[`*binder*`]`*term* means the Π dependend product. These constructions are extended to lists of variables.

  If x does not occurr free in B, the term `(x:A)B` can be written as `A->B`.

- The `Case` represents a case analysis over a term of an inductive type.

- The syntax `Fix` is used for the internal representation of fixpoints.

The object logic is encoded by *declaring* in Coq its language and proof rule. A *declaration* is the association between an *ident* and a *term*. A declaration is accepted by Coq iff this *term* is a well-typed specification in the current context of the declaration, and

$$
\begin{array}{rcl}
\mathit{term} & ::= & \mathit{ident} \mid \mathit{sort} \mid (\mathit{binder})\mathit{term} \mid [\mathit{binder}]\mathit{term} \mid \\
& & (\mathit{terms}) \mid \texttt{<}\mathit{term}\texttt{>}\ \texttt{Case}\ \mathit{term}\ \texttt{of}\ \mathit{terms}\ \texttt{end} \\
& & \texttt{Fix}\ \mathit{ident}\ \{\mathit{fixdecls}\} \\
\mathit{ident} & ::= & (\texttt{a..z} \mid \texttt{A..Z} \mid \texttt{\_} \mid \$)\{\texttt{a..z} \mid \texttt{A..Z} \mid \texttt{\_} \mid \$ \mid ')^{+} \\
\mathit{terms} & ::= & \mathit{term} \mid \mathit{term}\ \mathit{terms} \\
\mathit{binder} & ::= & \mathit{lident}\ :\ \mathit{term} \\
\mathit{lident} & ::= & \mathit{ident} \mid \mathit{ident},\ \mathit{lident} \\
\mathit{sort} & ::= & \texttt{Prop} \mid \texttt{Set} \mid \texttt{Type} \\
\mathit{fixdecls} & ::= & \mathit{fixdecl} \mid \mathit{fixdecl}\ \texttt{with}\ \mathit{fixdecls} \\
\mathit{fixdecl} & ::= & \mathit{ident}\ /\ \mathit{num}\ :\ \mathit{term}
\end{array}
$$

Figure 10.4: (A fragment of) The Gallina specification language.

ident is not previously declared. The two principal means for declaring an identifiers are the following:

`Axiom` *ident* `:` *term*.
`Variable` *ident* `:` *term*.

In both cases, the *term* is taken to be the type of *ident*.

Inductive types are declared by the syntax

$$
\begin{array}{rcl}
\texttt{Inductive}\ \mathit{ident}\ :\ \mathit{term}\ & := & \mathit{ident}_1\ :\ \mathit{term}_1 \\
& \mid & \ldots \\
& \mid & \mathit{ident}_n\ :\ \mathit{term}_n.
\end{array}
$$

The name *ident* is the name of the inductively defined object, and *term* is its type. The names $\mathit{ident}_1, \ldots, \mathit{ident}_n$ are the names of its constructors and $\mathit{term}_1, \ldots, \mathit{term}_n$ their respective types. The types of the constructors, as in CIC, have to sarisfy a *positivity condition* (see [CCF+95, section 6.5.3]).

10.4.2 The proof assistant

After having declared the object logic, the user can carry out formally proofs.

At each stage of a proof development, Coq mantains a list of goals to prove. Initially, the list consists only in the theorem itself, which can be declared by the command `Theorem` *ident* `:` *term*. This command switches Coq in proof editing mode, and sets *term* as the original goal. At each stage, each goal is provided by a number of available hypothesis which form the *local proof context* of the goal. Initially, this context is empty; it is enriched by the use of *introduction tactics*. Of course, assumptions can be used in proving a goal.

In Coq's proof editing mode, the user has access to specialized commands (*tactics*) dealing with proof development. There are three levels of tactics. The simplest one

implements the basic rules of the logical framework, such as `Intro` which implement the ABS rule of PTS's. The second level is the one of *derived rules*, which are built by combination of other tactics; several conversion rules belong to this level. The third level is the one of *heuristic* or *decision* procedures for building a complete proof of a goal (see e.g. the `Auto` tactic).

When a proof is achieved, the message `Subtree proved!` is displayed. One can then store this proof as a defined constant in the environment. The theorem can be used later on in other proof developments.

Remark. In the next chapters, we will use CIC in several examples and occasions. We will use both a pretty-printed version of the pre-λ-terms and the real language used in the Coq implementation of CIC. The last one will be written in `verbatim`.

10.5 Paradigmatical *Judgement-as-Types* encodings

In this section, we formalize the notion of "adequate" and "natural" encoding of a formal system. We are inspired by [HHP93, Gar92, Gar93].

We begin with the formalization of a "good representation" of a language:

Definition 10.5 (Encoding of a language) *Let $\mathcal{L}$ be a language and let Sorts, Expr the sorts and expressions of $\mathcal{L}$, respectively. Let X, Y range over finite lists of (sorted) variables of the language, that is sets of the form $\{x_1^{S_1}, \ldots, x_n^{S_n}\}$.*

An encoding *of $\mathcal{L}$ in CIC is a triple $(\Sigma, \eta, \varepsilon)$ where Σ is a signature, $\eta : Sorts \to \Lambda$ is a function such that $\forall S \in Sorts : \emptyset \vdash_\Sigma \eta(S) : \mathsf{Set}$ and $\eta(S)$ is in canonical form; ε is a family of functions indexed by lists of variables, $\varepsilon_X : Expr_X \to \Lambda$, such that for $X = \{x_1^{S_1}, \ldots, x_n^{S_n}\}$, we have:*

1. *for $x^S \in X$, let $\varepsilon_X(x) = x$ where the right-hand side "x" is a variable of CIC of sort $\eta(S)$; let then*
$$\Xi(X) \stackrel{\text{def}}{=} \langle \varepsilon_X(x_1) : \eta(S_1), \ldots, \varepsilon_X(x_n) : \eta(S_n) \rangle$$
2. *for each sort $S \in Sorts$ and expression $e \in S_X$, we have $\Xi(X) \vdash_\Sigma \varepsilon_X(e) : \eta(S)$, and $\varepsilon_X(e)$ is in canonical form;*
3. *ε_X is compositional, that is for every $e_1, \ldots, e_m \in Expr_X$, and for every set of fresh variables $Y = \{y_1, \ldots, y_m\}$, for every expression $e \in Expr_{X,Y}$:*
$$\varepsilon_X(e[e_1/y_1, \ldots, e_m/y_m]) = (\varepsilon_{X,Y}(e))\,[\varepsilon_X(e_1)/\varepsilon_Y(y_1), \ldots, \varepsilon_X(e_m)/\varepsilon_Y(y_m)]$$

The encoding is adequate *if the maps are bijections onto the sets of canonical forms:*

1. *the function $\eta : Sorts \to \{A \mid \emptyset \vdash_\Sigma A : \mathsf{Sort},\ A \text{ canonical}\}$ is a bijection;*
2. *for $X = \{x_1^{S_1}, \ldots, x_n^{S_n}\}$ and for $S \in Sorts$, the function $\varepsilon_X : S_X \to \{M \mid \Xi(X) \vdash_\Sigma M : \eta(S), M \text{ canonical}\}$ is a bijection.*

Now we turn to the representation of proof systems. Differently from other approaches [HHP93, Gar92, Gar93], we focus on multiple consequence relations with possibly infinite assumptions.

Proof-irrelevant encodings

There are (at least) two notions of "satisfying representation" of proof system. The first approach is to take the approach of *proof irrelevance*: we do not care of reflecting faithfully the derivations of the proof systems in the signature; instead, we just require the *existence* of a witness for each derivation [Gar93]. In other words, what we represent is simply the (multiple) consequence relation defined by the proof system.

Definition 10.6 (Proof-irrelevant encoding) *Let $\mathcal{L}$ be a language, and $(\Sigma, \eta, \varepsilon)$ its encoding. Let $(\rhd_i)_{i\in I}$ a multiple consequence relation on $\mathcal{L}$. The* (proof-irrelevant) encoding *of $(\rhd_i)_{i\in I}$ is a pair (Σ', J) where*

- *Σ' is a signature extending Σ, and*
- *J is a family of preterms $(J_i)_{i\in I}$,*

such that, if $\Gamma_1; \ldots; \Gamma_n \rhd_i \varphi$ with variables in X then

1. *$\Xi(X) \vdash_{\Sigma'} (J_i\ \varepsilon_X(\varphi)) : \mathsf{Prop}$ and it is in canonical form;*
2. *let $\gamma^i_X(\Gamma_i)$ the* encoding of Γ_i*, defined as follows:*
   - *if Γ_i is finite, then*

$$\begin{array}{rcl} \gamma^i_X(\emptyset) & \stackrel{\text{def}}{=} & \emptyset \\ \gamma^i_X(\Gamma, \psi) & \stackrel{\text{def}}{=} & \gamma^i_X(\Gamma), a : (J_i\ \varepsilon_X(\psi)) \end{array}$$

     where the variable a is fresh.
   - *otherwise, let Γ_i be infinite (but still r.e.); let φ_n for $n \in \mathcal{N}$ an enumeration of Γ_i. Then there is a λ-term G such that $\forall n \in \mathcal{N} : (G\ \bar{n}) = \varepsilon_X(\varphi_n)$. Then, $\gamma^i_X(\Gamma_i) \stackrel{\text{def}}{=} (a : \Pi n{:}nat.(J_i\ (G\ n)))$.*
3. *exists M such that $\Xi(X), \gamma^1_X(\Gamma_1), \ldots, \gamma^n_X(\Gamma_n) \vdash_{\Sigma'} M : (J_i\ \varepsilon_X(\varphi))$.*

The encoding is adequate *if the interpretation is* complete*, in the following sense: for $X = \{x_1^{S_1}, \ldots, x_n^{S_n}\}$, $(\Gamma_i)_{i\in I}$ family of contexts and φ formula with free variables in X, if there exists M such that $\Xi(X), \gamma^1_X(\Gamma_1), \ldots, \gamma^n_X(\Gamma_n) \vdash_{\Sigma,\Sigma'} M : (J_i\ \varepsilon_X(\varphi))$, then $\Gamma_1; \ldots; \Gamma_n \rhd_i \varphi$.*

Remark. In the above definition, the representation of infinite sets of assumptions relies on the existence of a λ-term enumerating an infinite (but r.e.) set. This is not ensured in every Logical Framework, because the underlying typed λ-calculus may have not a sufficient expressive power. This is the case, for instance, of the Edinburg LF and ELF$^+$ [HHP93, Gar92]. Instead, any typed λ-calculi extending Girard's F [GLT90] (such as the Calculus of Inductive Constructions) admit this construction.

Of course, finite contexts could be represented also by "infinitary" tools:

Proposition 10.11 *Let $\Gamma = \{\varphi_0, \ldots, \varphi_n\}$ be a non-empty finite set of formulæ with variables in X. Then, there is a λ-term G such that*

$$\forall i = 0, \ldots, n: \quad (G\ \bar{i}) = \varepsilon_X(\varphi_i)$$
$$\forall i > n: \quad (G\ \bar{i}) = \varepsilon_X(\varphi_n)$$

Proof. Just consider that Γ can be described as the sequence $\{\varphi_1, \ldots, \varphi_n, \ldots\}$, where for $j \geq n : \varphi_j = \varphi_n$.

Of course, in the case of finitary contexts, the finitary representation (that is, by means of a constant for each assumption) is easier to handle and clearer than the infinitary one.

From a proof-irrelevant perspective, therefore, a logic can be viewed as a language together with a multiple consequence relation; hence, the encoding of a logic is just the encoding of the language together with the encoding of the MCR:

Definition 10.7 (Proof-Irrelevant Encoding of a Logic) *Let $LOG = (\mathcal{L}, \rhd)$ be a logic. An* encoding *of LOG is a quadruple $\langle \Sigma, \eta, \varepsilon, J \rangle$ such that*

- *Σ is a signature;*
- *$\langle \Sigma, \eta, \varepsilon \rangle$ is an encoding for $\mathcal{L}$;*
- *$\langle \Sigma, J \rangle$ is an encoding for $\rhd$.*

Natural encodings

Let us consider the case in which we require also a "good" representation of the proof systems. Following the *judgement-as-types, proof-as-λ-terms* paradigm, we usually require also a correspondence between proofs of a consequence relation, and terms inhabiting the corresponding judgement. Therefore, we extend the previous definition of encoding, in order to capture also this correspondence.

Some notational remarks. In the following, for typographical reason, we will denote by Γ a family of sets of assumptions $(\Gamma_i)_i$, and by $\gamma_X(\Gamma)$ its encoding $\gamma^1_X(\Gamma_1), \ldots, \gamma^n_X(\Gamma_n)$.

Moreover, we denote by $\pi : \Gamma \vdash_i \varphi$ a proof π (in a given system S) that φ is an ith consequence of the (family of) set of assumptions Γ (We suppose that these proofs are represented in some finitary way, e.g. by labelled trees or proof expressions [HHP93]).

For $\pi : (\Gamma \vdash_i \varphi)$ in X, e an expression and $x \in X$, we denote by $\pi[e/x]$ the proof obtained from π by the standard substitution of e for x (and hence, it is $\pi[e/x] : (\Gamma[e/x] \vdash_i \varphi[e/x])$). Moreover, if σ is a proof on Γ, φ, we denote by $\sigma[\pi/\varphi]$ the proof obtained by componing the proof π on σ in the assumption φ (i.e. a "plugging" in the style of Natural Deduction).

We define then $Proofs^i_{X,\Gamma,\varphi} \stackrel{\text{def}}{=} \{\pi \mid \pi : (\Gamma \vdash_i \varphi), \mathrm{FV}(\pi) \subseteq X\}$.

Definition 10.8 (Natural encoding) *Let $\mathcal{L}$ be a language, and $(\Sigma, \eta, \varepsilon)$ its encoding. Let S be a proof system on $\mathcal{L}$, defining the (possibly) multiple consequence relation $(\vdash_i)_i$.*

A natural encoding *of S is a triple (Σ', J, ξ) where (Σ', J) is a proof-irrelevant encoding of $(\vdash_i)_i$ and ξ is a family of functions, such that for all X, Γ, φ:*

$$\xi^i_{X,\Gamma,\varphi} : Proofs^i_{X,\Gamma,\varphi} \to \Lambda$$

and the following properties hold:

1. *if* $\pi : \Gamma_1; \ldots; \Gamma_n \vdash_i \varphi$ *with variables in* X*, then*

$$\Xi(X), \gamma_X^1(\Gamma_1), \ldots, \gamma_X^n(\Gamma_n) \vdash_{\Sigma'} \xi^i_{X,\Gamma,\varphi}(\pi) : (J_i\ \varepsilon_X(\varphi))$$

2. *let* $\pi : \Gamma_1; \ldots; \Gamma_n \vdash_i \varphi$ *be a trivial proof of an assumption* $\varphi \in \Gamma_j$*. If* Γ_j *is finite, then* $\xi^i_{X,\Gamma,\varphi}(\pi)$ *is a variable in* $\mathrm{dom}(\gamma^j_X(\Gamma_j))$*. Otherwise, let* φ *be the* n*th formula in the enumeration of* Γ_j*; then,* $\xi^i_{X,\Gamma,\varphi}(\pi)$ *is* $(a\ \bar{n})$*, where* $a \in \mathrm{dom}(\gamma^i_X(\Gamma))$

3. *each component of* ξ *is compositional on expressions; that is, for each sort* S*, list of variables* (X, x^S)*, expression* $e \in S_X$*, if* $\pi : \Gamma \vdash_i \varphi$ *is a proof in* (X, x)*; then*

$$\xi^i_{X,\Gamma,\varphi}(\pi[e/x]) = (\xi^i_{(X,x^S),\Gamma,\varphi}(\pi))[\varepsilon_X(e)/x]]$$

4. *each component of* ξ *is compositional on proofs; that is, for each* $\Gamma = (\Gamma_i)_i$ *and* $\Gamma' = (\Gamma_j)_{j<i}; \Gamma_i, \varphi; (\Gamma_j)_{j>i}$*, if* $\pi : \Gamma \vdash_i \varphi$ *and* $\sigma : \Gamma' \vdash_k \psi$ *in* X*, then*

$$\xi^k_{X,\Gamma,\psi}(\sigma[\pi/\varphi]) = (\xi^k_{X,\Gamma',\psi}(\sigma))[\xi^i_{X,\Gamma,\varphi}(\pi)/\xi^i_{X,\Gamma,\varphi}(\varphi)]$$

The encoding is adequate *if* ξ *is* complete*, in the sense that the restriction*

$$\xi^i_{X,\Gamma,\varphi} : \mathit{Proofs}^i_{X,\Gamma,\varphi} \to \{t \mid \Xi(X), \gamma(\Gamma) \vdash t : (J_i\ \varepsilon_X(\varphi)), t \text{ canonical}\}$$

is a bijection, such that for every proof π*:*

$$\Xi(X), \gamma(\Gamma) \vdash \xi^i_{X,\Gamma,\varphi} : (J_i\ \varepsilon_X(\varphi)) \quad \Rightarrow \quad \pi : \Gamma \vdash_i \pi$$

Finally, we can give the definition of an adequate and natural encoding of a proof system:

Definition 10.9 (Encoding of a proof system) *Let* $\mathcal{L}$ *be a language, and* S *a proof system on* $\mathcal{L}$*. An* adequate encoding *of* S *is a 5-tuple* $\langle \Sigma, \eta, \varepsilon, J, \xi \rangle$ *such that*

- Σ *is a signature;*
- $\langle \Sigma, \eta, \varepsilon \rangle$ *is an encoding of* $\mathcal{L}$*;*
- $\langle \Sigma, J, \xi \rangle$ *is an adequate encoding of* S*.*

The encoding is natural *if* ξ *is complete.*

It is worthwhile stressing that the encoding maps are fundamental components of any encoding. The activity of encoding a logic can be summarized in the following general schema [HHP93, Gar92]:

Judgements-As-Types Encoding Paradigm

1. define a signature for the encoding of the syntax; for each sort introduce a distinct type, and for each expression constructor a distinct constant of the right type, obtained by "currying" the arity of the expression constructor.

2. define the encoding maps η, ε;
3. prove their adequacy (i.e. prove the *adequacy theorem for syntax*, corresponding to [HHP93, Theorem 3.1.1]);
4. define a signature for the encoding of the system; for each consequence relation introduce a distinct judgement, and for each inference rule introduce a distinct constant of the right type.
5. define the encoding map for proofs, ξ;
6. prove its adequacy (i.e. prove the *adequacy theorem for proofs* corresponding to [HHP93, Theorem 4.1.1]).

Chapter 11

Encoding of Formal Systems in the Calculus of Inductive Constructions

The standard "judgement-as-types" paradigm of representation presented in the previous chapter, is strictly related to Martin-Löf's theory of arities and judgements. It has been designed for taking full advantages of the metalogical features offered by the Logical Framework, delegating as much as possible to the metalanguage [HHP93, AHMP92]. The methodology can be summarized in three steps:

- each syntactic sort is represented by a type inhabiting Set;
- variables of a syntactic sort are represented by variables of the corresponding type (*higher-order abstract syntax*);
- each basic judgement is represented by a type inhabiting Prop.

The multi-sorted, higher-order abstract syntax paradigm is a clean and successful approach to the representation of formal systems. Adopting this technique, we delegate to the metalanguage many complex features of the object language:

- type-checking is delegated to the typing system of the metalanguage;
- object variables are represented by metalogical variables;
- α-conversion is enforced by the α-conversion of the metalanguage;
- substitution is delegated to the substitution of the metalanguage;

However, this approach is not always feasible. Representing object variables with variable of the metalanguage yields precise connections between the metalogical properties of object logic and the structural properties of the metalanguage. As we will see in Section 11.1, in the case of type-theory based Logical Frameworks this means that precise closure properties of the language and schematicity of the proof system are required, in order to have an adequate HOAS representations. Unfortunately, most languages and program

logics do not enjoy these properties (e.g. Dynamic Logic (cf. Section 3.2.3), Hoare Logic, etc.); in these cases, a direct HOAS is not feasible.

Another source of complication is the need of inductions principles on the language. Most proofs on programs are carried out by induction on the syntax of programs, terms, expressions and other syntactic objects; hence, induction principles are highly advisable. One of the reasons for adopting the Calculus of *Inductive* Construction is that it can provide, automatically, powerful induction/recursion principles (see Section 10.3). However, the inductive definitions of CIC are not feasible in presence of syntactic constructors of negative arity. This is the case, indeed, of many program languages and logics, such as the λ-calculus and the μ-calculus.

Therefore, in implementing a program logic, one has to face several design choices in order to reach a good trade-off between the possibility of delegating complex features to the metalanguage, and brute force encodings. This is the object of this chapter.

It turns out that α-conversion is *always* delegable to the metalanguage, while the substitution may need to be implemented "by hand". This is not so surprising, indeed, since the theory of arities offers an uniform treatment only for the α-conversion, and nothing is said about substitution (see Chapter 2).

In this chapter, we discuss these nonparadigmatical situations. We streamline some of the techniques and "tricks" appearing in the literature [BH90, AHMP92, HHP93, DH94, Mic94, DFH95, Ter95].

We proceed as follows. In Sections 11.1 and 11.2, we will examine two general situation in which the HOAS clashes with metalogical properties of the system we are encoding. In Section 11.1 we examine in detail the metalogical properties the object logic has to enjoy in order to be adequately encodable.

In Section 11.2 we investigate the applicability of the inductive definitions offered by CIC. We will examine both first-order and higher-order definitions (Section 11.2.1, 11.2.2). Despite their generality, the former are not always satisfactory, e.g. when we deal with binding constructors; on the other hand, unfortunately, higher-order inductive abstract syntax definitions are not always feasible. In these unfortunate cases, one may choose among three possible solutions, which we outline in Section 11.2.3.

The probably most promising solution is to delegate only the α-conversion to the metalanguage, and to encode the substitution mechanism "by hand". Therefore, an investigation of efficient encodings of substitutions is in order. This investigation is carried out in Section 11.3, where we discuss two main different approaches to the implementation of substitution. The first is to implement faithfully the textual, syntactic notion of substitution (adopted e.g. in [DFH95]); the second is to adopt an approach closer to the "semantical" meaning of substitution, implementing the bookkeeping technique presented in Section 4.2.2.

Finally, in Section 11.4 we briefly discuss two further situations which are still difficultly dealt with by the presented encoding paradigm in actual Logical Frameworks. These situations are when there are "context-sensitive" conditions on the syntactic constructors (such as in context-sensitive grammars), and when there are subsorts. In the former case, we propose also an extension of the Calculus of Inductive Constructions, in order to augment its applicability range.

In this chapter we do not focus on advanced techniques for representing peculiarities of proof systems, such as modalites, proof rules, infinitary rules and so on. These features

will be discussed in the following chapters.

We refer also to Appendix D, where we summarize the main results of this chapter in a "roadmap", which should guide the choice of the right methodology for encoding the language of a given formal system.

11.1 Schematicity induced by HOAS

Intuitively, it is clear that in order to obtain an adequate encoding of a given logic, the metatheory of the logic has to be compatible with the metatheory of the metalanguage.

As we have seen, object-logic variables ranging over a sort S are represented by means of metalogical variables of the type corresponding to S. These meta-logical variables behave accordingly to the rules of the typed λ-calculus adopted as LF. In particular, a variable acts as a *placeholder* for terms of the same type. Hence, an adequate representation reflects this property of metalogical variables in the object-logic, leading to precise closure properties of both the syntax and logical systems. In other words, an adequate represention is feasible only if the logic enjoys some closure properties, as shown in the following results.

Lemma 11.1 *Let $\mathcal{L}$ be a language, adequately represented by $\langle \Sigma, \eta, \varepsilon \rangle$.*

Let $X = \{x_1^{S_1}, \ldots, x_n^{S_n}\}$ be a set of variables, x_i ranging over sort S_i. Let S_0 be a syntactic sort, represented by the kind $\eta(S_0) = \theta_0$ and a compositional bijection $\varepsilon_{0X} : S_{0X} \to \{M \mid \Xi(X) \vdash M : \theta\}$; let δ_{0X} be the inverse of ε_{0X}. Let $x^{S_0} \notin X$ be a variable ranging over S_0.

Let S be a sort (which can depend on X, x), represented by a type θ and a compositional bijection $\varepsilon_{X,x} : S_{X,x} \to \{M \mid \Xi(X), x : \theta_0 \vdash M : \theta\}$.

Then, for all $e \in S_{X,x}$ and r canonical such that $\Xi(X) \vdash r : \theta_0$, we have $e[\delta_{0X}(r)/x] \in S[\delta_{0X}(r)/x](X)$.

Proof. Let e, r be as in the hypothesis; then, $\Xi(X), x : \theta_0 \vdash \varepsilon_{X,x}(e) : \theta$. By the Substitution Lemma (Theorem 10.1), we have $\Xi(X) \vdash \varepsilon_{X,x}(e)[r/x] : \theta_1[r/x]$. By compositionality of $\varepsilon, \varepsilon^0$, this is equivalent to $\Xi(X) \vdash \varepsilon_X(e[\delta_X^0(r)/x]) : \theta_1[r/x]$, where the substitution $e[\delta_X^0(r)/x]$ exists for the composition bijectivity of ε_X. By bijectivity, this means also that $e[\delta_X^0(r)/x] \in S[\delta_X^0(r)/x](X)$. □

This lemma has two main implications. The first concerns the representation of the syntax. Roughly, if a language is adequately represented in a Logical Framework, then it is closed under instantiation of the variables. The second is a similar closure of the inference system of the logic. Roughly, if the language is adequately represented and we can deduce a judgement J from a set of assumptions Γ, then for every free variable x in Γ, J, we can deduce also $J[t/x]$ from $\Gamma[t/x]$. These statements are formalized by the following result.

Theorem 11.2 *Let $(\mathcal{L}, \rhd)$ be a logic adequately represented by $\langle \Sigma, \eta, \varepsilon, J \rangle$. Let $X = \{x_1^{S_1}, \ldots, x_n^{S_n}\}$ be a set of variables, x_i ranging over the sort S_i; let S_0 a syntactic sort, represented by a type θ_0 and a compositional bijection $\varepsilon_X^0 : S_{0X} \to \{t \mid \Xi(X) \vdash t : \theta_0\}$. Let $x^{S_0} \notin X$ be a variable ranging over S_0, and $e_0 \in S_{0X}$.*

1. *Let S be a syntactic sort represented by a type $\theta : \mathsf{Sort}$ and a compositional bijection $\varepsilon_Y : S_Y \to \{t \mid \Xi(Y) \vdash t : \theta\}$. Then, for all $e \in S_{X,x} : e[e_0/x] \in S_X$.*

2. *Let* $\Gamma_1; \ldots; \Gamma_n \rhd_i \varphi$ *be a consequence with variables in* X, x. *Then,* $\Gamma_1[e_0/x]; \ldots; \Gamma_n[e_0/x] \rhd_i \varphi[e_0/x]$ *holds.*

Proof. The first statement is a straightforward consequence of Lemma 11.1; just take $r \stackrel{\text{def}}{=} \varepsilon_{0X}(e_0)$. The second is similar: if $\Gamma_1; \ldots; \Gamma_n \rhd_i \varphi$ then there is a term M such that

$$\Xi(X, x), \gamma^1_{X,x}(\Gamma_1), \ldots, \gamma^n_{X,x}(\Gamma_n) \vdash_\Sigma M : (J_i\ \varepsilon_{X,x}(\varphi)).$$

By abstracting over the variables representing the assumptions, we obtain

$$\begin{array}{rcl} \Xi(X), x : \theta_0 & \vdash & \lambda\overline{a_1}{:}\gamma_{X,x}(\Gamma_1) \ldots \lambda\overline{a_n}{:}\gamma_{X,x}(\Gamma_n).M \\ & & : \Pi\overline{a_1}{:}\gamma_{X,x}(\Gamma_1) \ldots \Pi\overline{a_n}{:}\gamma_{X,x}(\Gamma_n).(J_i\ \varepsilon_{X,x}(\varphi)) \end{array}$$

and hence, for the lemma,

$$\begin{array}{rcl} \Xi(X) & \vdash & (\lambda\overline{a_1}{:}\gamma_{X,x}(\Gamma_1) \ldots \lambda\overline{a_n}{:}\gamma_{X,x}(\Gamma_n).M)\,[\varepsilon_{0X}(e_0)/x] \\ & & : (\Pi\overline{a_1}{:}\gamma_{X,x}(\Gamma_1) \ldots \Pi\overline{a_n}{:}\gamma_{X,x}(\Gamma_n).(J_i\ \varepsilon_{X,x}(\varphi)))\,[\varepsilon_{0X}(e_0)/x]\,. \end{array}$$

Introducing the variables back in the context, we obtain

$$\begin{array}{rcl} \Xi(X), \gamma^1_X(\Gamma_1)[\varepsilon_{0X}(e_0)/x], \ldots, \gamma^n_X(\Gamma_n)[\varepsilon_{0X}(e_0)/x] & \vdash_\Sigma & M[\varepsilon_{0X}(e_0)/x] \\ & & : (J_i\ \varepsilon_X(\varphi)[\varepsilon_{0X}(e_0)/x]). \end{array}$$

By the adequacy of representation, the existence of the term $M[\varepsilon_{0X}(e_0)/x]$ corresponds to the validity of $\Gamma_1[e_0/x], \ldots, \Gamma_n[e_0/x] \rhd_i \varphi[e_0/x]$, which is the thesis. □

The second statement of the theorem can be restated as the admissibility of the *substitution rule*:

$$\frac{\Gamma_1; \ldots; \Gamma_n \rhd_i \varphi}{\Gamma_1[e_0/x]; \ldots; \Gamma_n[e_0/x] \rhd_i \varphi[e_0/x]} i \in I$$

that is, the multiple consequence relation $(\rhd_i)_{i\in I}$ has to be *schematic* with respect to the standard substitution (nondeterministic) schema of instantiation of free variables (Section 3.2).

Corollary 11.3 *Adequately represented languages are closed under substitution. Adequately represented proof systems admit the substitution rule.*

Therefore, logics which do not satisfy these schematicity are incompatible with the structural properties of the metalanguage, and hence cannot be naïvely represented by HOAS. These logics are not so rare as one may think.

Example 11.1 In Section 3.2.3 we have already seen a language which is not closed under standard substitution, namely the language of Dynamic Logic (Chapter 7). For instance, we cannot replace x by 1 in $[x := 0](x = 0)$, because we would obtain $[1 := 0](x = 0)$ which is not a formula of Dynamic Logic.

This problem arises whenever we face program logics with "program" variables which are used also as logical variables (See also the encoding of Hoare Logic in [AHMP92]). The encoding of Dynamic Logic will be discussed in Chapter 14. □

Example 11.2 We show a logic whose language is closed under standard substitution, but it proof systems does not admit the substitution rule. Consider any proof system for the FOL of validity, that is, a proof system representing the $\models$ defined in Definition 3.4 (such a system is, for instance, the usual Hilbert-style system with the generalization rule).

Obviously, for every φ, t and x, $\varphi[t/x]$ is a formula of FOL. However, validity FOL does not admit the substitution rule, since the sequent $x = 0 \vdash \forall x.x = 0$ is valid while $0 = 0 \vdash \forall x.x = 0$ is not. Hence, the substitution notion in validity FOL cannot be delegated by means of HOAS to the metalanguage. (See App.C for some Coq code). □

Our conjecture is that this condition is also *sufficient*.

Conjecture 11.1 *Let $(\mathcal{L}, \rhd)$ be a logic,*[1] *S a sort with variables. Variables of S can be represented by metalogical variables iff*

- *each sort of $\mathcal{L}$ is closed with respect to standard substitution;*
- *the (multiple) consequence relation $\rhd$ admits the substitution rule over S.*

In the case the logic does not satisfy these closure condition, we can still delegate to the metalanguage the α-conversion of bound variables, but we cannot delegate the substitution mechanism. This can be achieved by introducing a specific set for the variables, and changing the arity of the binding constructor accordingly, as follows.

Let S a sort with variables, such that one of the above conditions fails (for instance, there is $e \in Expr_{X,x}$ but for $t \in S_X$ it is $e[t/x] \notin Expr_X$). Then,

1. we introduce a new sort, say V, with variables and no other constructor;
2. we add a new expression constructor v of arity $V \to S$ (called *coercion*), which embeds V in S.
3. in the arity of every constructor which binds a variable of S, we replace the bound occurrence S by V; e.g., the arity of $c : (S \to S_1) \to S$ becomes $(V \to S_1) \to S$.

After this transformation, the α-conversion of bound variables is still delegate to the metalanguage, but we cannot freely replace a variable of V by means of an expression of S. Therefore, we have to take care of implementing the "right" substitution schema "by hand", accordingly to the metatheoretic properties of the object logic. In Section 11.3 we will see some approaches to the implementation of substitution.

11.2 Feasibility of Inductive Definitions

One of the main reasons for adopting CIC as Logical Framework, is its capability of provides automatically (under certain conditions) powerful induction principles on sets and propositions 10.3. In the case of the encoding of the language, this is particular important, since many proofs of properties of programs are carried by structural induction on the syntax [Plo81]. In this section we investigate the applicability of the inductive definitions offered by CIC. We will examine both first-order and higher-order definitions (Section 11.2.1, 11.2.2). Despite their generality, the former are not always satisfactory,

[1] The consequence relation can be defined by a proof system, as well.

e.g. when we deal with binding constructors; on the other hand, unfortunately, higher-order inductive abstract syntax definitions are not always feasible. In these unfortunate cases, one may choose among three possible solutions, which are outlined in Section 11.2.3.

11.2.1 First-Order Inductive Abstract Syntax

Let us begin with the most simple case, that is, when the language have no binding constructors. In this case, we apply directly the standard encoding paradigm for declaring an inductive tupe, as follows.

Let $S \in \mathit{Sorts}$ be a sort, and c_i, for $i = 1, \ldots, m$, the constructors of sort S, whose arities are $(S_{i1}, \ldots, S_{in_i}) \to S$ respectively. Then, the sort S can be represented in CIC by the following definition:

$$\begin{array}{rrcl} \textbf{Inductive Set } S & := & & \\ & c_1 & : & S_{11} \to \ldots \to S_{1m_1} \to S \\ & & \vdots & \\ \mid & c_n & : & S_{n1} \to \ldots \to S_{nm_n} \to S. \end{array}$$

This definition automatically provides the induction and recursion principles. The case of mutually inductive languages is treated similarly, by virtue of the `Mutual Inductive` definitions of Coq.

In many cases, this solution is perfect – for instance, lists, trees, forests, numerals, propositional logic and every free algebra. Induction and recursion principles are automatically defined and proved by the system. Encoding bijections are easily defined by induction on the syntax.

Example 11.3 The language of lists of numerals:

$$\begin{array}{rrcl} \mathit{nat}: & n & ::= & 0 \mid Sn \\ \mathit{list}: & l & ::= & \langle\rangle \mid n.l \end{array}$$

is represented in Coq as follows:

```
Inductive nat : Set :=
          O : nat
        | S : nat -> nat.

Inductive list : Set :=
          empty : list
        | cons  : nat -> list -> list.
```

□

This solution is adequate and successful only if the language we encode has no binding operators. In fact, a naïve, first-order implementation of binding operators is very unsatisfactory. We *do not* delegate variables and α-conversion to the metalanguage, by adopting the higher-order abstract syntax; instead, we "flatten" the arity of every constructor, and then we define a free algebra.

Example 11.4 A trivial implementation of the λ-calculus.

```
Parameter var : Set.

Inductive L : Set :=
          ref : var -> L
        | lam : var -> L -> L
        | app : L -> L -> L.
```

Although $\lambda x.x \equiv \lambda y.y$, the terms `(lam x (ref x))` and `(lam y (ref y))` are completely different. We need to implement all the equational theory of terms. □

Of course, this approach is very unsatisfactory: it yields a *too* fine-grained representation, and it forces us to implement all the machinery of α-conversion and substitution.

In the next subsection we describe a more feasible approach.

11.2.2 Higher-order inductive abstract syntax

An advisable idea would be to follow the HOAS approach for defining inductive data types. Unfortunately, this is not always feasible. Due to consistency problems of the typing system of CIC, only strictly positive constructors are allowed by CIC in inductive definitions [CCF+95, Pau93]. Hence, if the sort S of the language we are encoding has a non-positive constructor, we cannot adopt the higher-order abstract syntax for defining an inductive type. Two examples are the λ-calculus and the μ-calculus.

Example 11.5 The following definitions of the syntax of λ-calculus and of the (pre)syntax of μ-calculus are rejected by Coq:

```
Inductive Set L :=
          app    : L -> L -> L
        | lambda : (L -> L) -> L.

Inductive Set o :=
          ff    : o
        | And   : o -> o -> o
        | mu    : (o -> o) -> o.
```

□

In the favourable situation there are no negative constructors, we can indeed apply the HOAS in an inductive definition.

Example 11.6 The encoding of First Order Logic.

```
Inductive Set i  :=
          zero   : i
        | succ   : i -> i
        | plus   : i -> i -> i.

Inductive Set o  :=
          equal  : i -> i -> o
        | And    : o -> o -> o
        | Forall : (i -> o) -> o.
```

□

Exotic Terms

The use of HOAS over inductive sets gives rise to another problem: the existence of extra terms which are not the representation of any expression of the object language, although they are in normal form and well-typed. The terms are called *exotic*, after [DFH95]. For instance, in the above representation of FOL, the term

```
(Forall [x:i](Eq zero
                <i>Case x of
                  (* zero *) zero
                  (* succ *) [y:i](succ i)
                  (* plus *) [y:i][z:i](plus y z)
                end)
```

is in head normal form and has type o, although it is not the encoding of any formula of FOL, although it is extensionally equivalent to the encoding of $\forall x.(0 = x)$. Beside these "extensionally acceptable" terms, there are also terms which are not even extensionally equivalent to any representation of formulæ, such as the following:

```
(Forall [x:i](Eq zero
                <i>Case x of
                  (* zero *) zero
                  (* succ *) [y:i]zero
                  (* plus *) [y:i][z:i](plus y z)
                end)
```

Exotic terms arise when there is a higher-order constructor (e.g. the `forall` above) which abstracts over an inductive type (e.g. the type of individuals above). In this situation, we can use a *Case* construction over an abstract variable, obtaining a head-normal form which do not correspond to any term.

These terms can be ruled out by introducing a "validity" judgement, which holds exactly on the *Case*-free terms. The definition of this judgement follows straightforwardly the syntax of terms[2]. We do not face here this problem; see [DH94, DFH95].

11.2.3 Possible solutions when Inductive HOAS fails

We have seen that the HOAS paradigm is not always compatible with inductive definitions. In this case, there are three solutions. The first idea is to drop the inductive definitions, and to apply the HOAS paradigm directly. On the opposite, we can look for a first-order formulation of the language, so that we can apply directly the inductive definitions without introducing higher-order construct. The third possibility is to try to combine HOAS and Inductive definitions.

Dropping the inductive definitions

We can always adopt the HOAS, dropping the induction principles. For instance, the λ-calculus can be defined as follows

[2]Some complications arise if we need to accept also exotic terms which are extensionally equivalent to valid ones; see [DFH95]

```
Parameter   L    : Set.
Parameter app    : L -> L -> L.
Parameter lambda : (L -> L) -> L.
```

but, of course, we lack any induction principle. This solution is a good alternative if we are not interested in the induction principles (or in frameworks which do not offer inductive definitions, such as the Edinburgh Logical Framework), but in the encoding of languages it is usually avoided.

Finding better first-order implementation: de Bruijn Notation

A well-known solution to the problem of α-conversion is the so-called *de Bruijn notation*. Using de Bruijn indexes there is no more α-conversion, hence we can adopt a straightforward first-order encoding of a free algebra. Indeed, in this setting the theory of λ-calculus becomes a theory of diadic trees of naturals.

This is de Bruijn representation of λ-terms [Hue92, DH94]:

```
Inductive lambda : Set :=
          Ref : nat -> lambda
        | Abs : lambda -> lambda
        | App : lambda -> lambda -> lambda.
```

There are no higher-order constructors: terms of type *lambda* are simple trees, whose leaves are labelled by naturals and internal nodes are labelled either by *Abs* or by *App*.

Hence, as far as the plain terms are concerned, a representation of binding operators by means of de Bruijn notation is faithful. Moreover, α-equivalence is immediately recovered. The drawback is that we cannot delegate substitution to the metalanguage: it has to implemented on its own. A complete formalization of the λ-calculus and the $\lambda\sigma$-calculus (λ-calculus with substitutions) has been carried out in Coq adopting this approach [Hue92, Hue94, Saï96].

In this setting, there are no exotic terms of type *lambda*, because we cannot create a object of type *lambda* containing a *Case* on an abstracted variable of type *lambda* [DFH95]. It is still possible create exotic terms at the level of *contexts*, that is λ-terms with holes. Contexts are represented by terms of type $lambda \to lambda$, and hence we can perform a *Case* on the abstracted variable:

```
Definition exotic1 : lambda -> lambda :=
  [x:lambda]<lambda>Case x of
                    [n:nat](Ref n)
                    [y:lambda](Abs y)
                    [y:lambda][z:lambda](App y z)
             end.

Definition exotic2 : lambda -> lambda :=
  [x:lambda]<lambda>Case x of
                    [n:nat](Ref (S n))
                    [y:lambda](Abs y)
                    [y:lambda][z:lambda](App y z)
             end.
```

The first term is extensionally equivalent to the "hole" context, $(\cdot)$, but not equal:

```
(* exotic1 and [x:lambda]x are extensionally equivalent,
   but they are not equal *)
Lemma exten_exotic1 : (y:lambda)(([x:lambda]x) y)=(exotic1 y).
Intro; Elim y; Intros; Reflexivity.
Qed.
```

On the other hand, $exot_2$ does not correspond to any context.

Combining Inductive Definition and HOAS

One, however, may try to delegating as much as possible by HOAS, still retaining the inductive definitions. Actually, the challence of combining HOAS and inductive principles is an ongoing area of research [DH94, DFH95, PW95, DPS96]. So far, the only solution feasible in CIC is to develop some special technique in order to eliminate the presence of negative constructors. A simple way for achieving this is to introduce a specific type for the identifiers.

Example 11.7 Acceptable definitions of λ-calculus and μ-calculus:

```
Parameter Var : Set.
(* here some axioms on Var *)

Inductive Set L :=
          app   : L -> L -> L
        | var   : Var -> L
        | lambda: (Var -> L) -> L.

Inductive Set o :=
          ff    : o
        | And   : o -> o -> o
        | var   : Var -> o
        | mu    : (Var -> o) -> o.
```

□

Therefore, in this way we can always delegate the α-conversion to the metalanguage, still obtaining an inductive definition. The unavoidable drawback of this approach is that the substitution is not delegated to the metalanguage, and hence it has to be implement "by hand". This is the subject of next section (Section 11.3).

11.3 Encoding of substitution schemata

Let us consider a case in which we cannot delegate the substitution to the metalanguage; as we have seen, this can be due to two reasons.

1. the clash between structural properties of the logic and those of the metalanguage (Section 11.1)

2. the presence of negative constructors and the need of inductive definitions (Section 11.2.2).

In both cases, there are two main ways for implementing substitution.

11.3.1 The syntax-oriented approach to substitution

The first approach is to implement an *inline* substitution, closely to the syntactic notion of substitution: a formalization of "textual replacement" of phrases for identifiers. This solution is feasible only when such a replacement makes sense (this is not the case, for instance, in *DL* and operational semantics; see Section 11.3.2 below).

This can be done by defining a judgement $subst_b : t_1 \to (Var \to t_2) \to t_3$ for each binding operator $b : (Var \to t_2) \to t_3$, where t_1 is the type representing the sort bound identifiers are supposed to range over. The intended meaning of $(subst\ A\ [x : Var]B\ C)$ is "C is obtained by replacing every occurrence of x in B by A". Rules of this judgement have to be introduced accordingly to the notion of instantiation we need. This approach has been studied in [DFH95, HM96].

Automated substitution and exotic terms

A natural question is: can *subst* be automated, defining it as a function $t_1 \to (Var \to t_2) \to t_3$, instead of a judgement as above? The answer is no, since it is defined by cases on the structure of B and the function space $Var \to t_2$ has no inductive structure, and hence no recursion principle is available on it.

The substitution can be automated if we define *subst* of type $t_1 \to Var \to t_2 \to t_3$; the meaning of $(subst\ A\ x\ B)$ is "the term obtained by replacing all free occurrences of x in B by A. In this case, we can apply the recursion principle on t_2 for defining *subst*, but intrisically we require here a recursive procedure for deciding when two identifiers are equal. This can be achieved only by instantiating Var over a particular recursive set, e.g. *nat*. In this case, we actually can define *subst* recursively as a function, but other problems arise about the existence of *exotic terms* (terms which are not representation of any real phrase), built by using the $Case$ constructor over Var.

Example 11.8 An exotic term in the encoding of λ-calculus

```
Definition var := nat.

Inductive L   : Set :=
          Var : var -> L
        | App : L -> L -> L
        | Lam : (var->L) -> L.

(* the following term does not represent a lambda term
    Lam [x:var](<L>Case x of
                   (* 0  *)   (Var 0)
                   (* Sn *)   [n:var](Var n)
                end)
*)
```

Suppose that there is a λ-term M corresponding to the above exotic term, and the nth identifiers i_n is represented by the nth numeral. Then $(M\ i_0) \rhd i_0$, while $(M\ i_{n+1}) \rhd i_n$, and this is absurd in the theory of λ-calculus. □

The solution is to specify a "validity" property, which holds only for "well-behaving" terms, ruling out the others. This judgement can be defined inductively on the structure of terms; see [DFH95] for more details.

11.3.2 A semantic-oriented approach to substitution

Substitution can be implemented also by the bookkeeping technique, described in Section 4.2.2. This takes full advantage of Natural deduction features of CIC (and more generally of type-theory based LF's). This approach is closer to the semantic notion of substitution and evaluation, and it can be applied also when a "standard" substitution is not feasible (such as in the case of program logics).

The basic idea is to do not perform the substitution; instead, the binding between the identifier and the substituting term is kept in the derivation context. The binding is mantained by an *ad-hoc* judgement. Recall the **let** evaluation rule of Section 4.2.2:

$$\frac{\begin{array}{c}(x' \mapsto n)\\ \vdots \\ N \Rightarrow n \quad M' \Rightarrow m'\end{array}}{\textbf{let}\ x = N\ \textbf{in}\ M \Rightarrow m}\ \mathsf{EC}(x, M, m)$$

where $\mathsf{EC}(x, M, m)$ means "M', m' are obtained from M, m respectively by replacing *all* the occurrences of x with x', which does not appears neither in x, M, m nor in any assumption different from $(x' \Rightarrow n)$".

The main difficulty in the implementation of bookkeepings is the enforcing of the side condition. Indeed, we need to implement the occur-checking of variables, by introducing two auxiliary judgements. This approach has been adopted, for instance, in the case of Natural Operational Semantics and Dynamic Logic [Mic94, HM96]; see Chapters 12, 14.

An alternative and interesting way for implementing the bookkeeping technique is to to take andvantage of the Leibnitz equality. This solution is particularly suitable when the object-level instantiation mechanism is the same of the metalevel one, but it cannot be delegated due to, say, negative binding constructors in inductive definitions (e.g. theory of λ-calculus, μ-calculus, recursive processes, etc.). Indeed, we can take advantage of the elimination principle of Leibnitz equality in an impredicative setting:

$$\mathit{eq\_ind} : \prod_{A:Set} \prod_{x:A} \prod_{P:A\to Prop} (P\ x) \to \prod_{y:A} x = y \to (P\ y)$$

This allows us to replace the occurrences of the bound variable with the passed expression. In Coq, applications of this principle are implemented by a simple tactic, `Rewrite`, which textually replaces terms for terms, accordingly to rules of the call-by-name λ-calculus.

Example 11.9 The encoding of untyped λ-calculus in Coq:

```
Parameter var : Set.
```

```
Inductive L   : Set :=
          Var : var -> L
        | App : L -> L -> L
        | Lam : (var->L) -> L.

(* call by name evaluation *)
Inductive eval : L -> L -> Prop :=
          eval_var : (x:var)(m:L)(Var x)=m -> (eval (Var x) m)
        | eval_lam : (M:var->L)(eval (Lam M) (Lam M))
        | eval_app : (B,n:L)(M:var->L)
                     ((x:var)(Var x)=B -> (eval (M x) n))
                     -> (eval (App (Lam M) B) n).

(* call by value evaluation *)
 Inductive evalv : L -> L -> Prop :=
          evalv_var : (x:var)(m:L)(Var x)=m -> (evalv (Var x) m)
        | evalv_lam : (M:var->L)(evalv (Lam M) (Lam M))
        | evalv_app : (A,B,n,m:L)(M:var->L)
                     (evalv A (Lam M)) ->
                     (evalv B m) ->
                     ((x:var)(Var x)=m -> (evalv (M x) n))
                     -> (evalv (App A B) n).

(* an example derivation *)
Lemma identity : (m:var->L)(eval (App (Lam [x:var](Var x)) (Lam m))
                                 (Lam m)).
Intro.
Apply eval_app.
Intros.
Rewrite -> H.
Apply eval_lam.
Qed.
```

□

This solution has been adopted, for instance, in the implementation of the μ-calculus; see Chapter 15.

11.4 Other complex situations

In this section we sketch briefly other situation in which the standard encoding paradigm is not satisfactory.

11.4.1 Conditioned Grammars

When one (or more) syntactic constructor is subject to formation conditions, we cannot adopt directly the methodology defined above.

Take a BNF description of a language L as follows:

$$A ::= c_1(A_{11}, \ldots, A_{1m_1}) \mid \ldots \mid c_n(A_{n1}, \ldots, A_{nm_n})$$

where c_i is a syntactic constructor (in prefix notation) of arity m_i and $A_{i1}, \ldots, A_{im_i}$ are non-terminal symbols of sorts $L_{i1}, \ldots, L_{im_i}$ respectively.

Constructor c_i is said to be *conditioned (by P)* if its application is subject to a condition $P(A_{i1}, \ldots, A_{im_i})$ of the arguments of c_i. A BNF-grammar is said to be *conditioned* if its constructors are conditioned.

A simple example of these languages is the language of lists of different naturals:

$$\begin{array}{lll} \mathit{DiffNats} : l & ::= & \langle\rangle \\ & & \mid l.n \quad \text{provided that } n \notin l \end{array}$$

This kind of condition is very common in Logic and programming languages (often it is said *static semantics* [Plo81]). For instance, in the μ-calculus, the formation of $\mu x.\varphi$ is subject to the condition that x occurs only positively in φ. In the λ_I-calculus, abstractions can be made only on really free variables. In many process algebra (CCS, WPA for instance), recursive processes have to be guarded. In Pascal, Algol, C and every other modern languages, a variable can appear only inside the scope of its declaration.

These features are common even in Proof Theory. For instance, in most implementations of Modal Logics, the introduction of $\Box$ relies on conditions on proofs (terms), which can be defined together with the notion of proof itself (see Chapter 13, the "Closed"-judgement and "Boxed"-judgement techniques).

A natural question arises: which is the class of conditioned grammars? The answer is: it depends on the expressive power of the conditions language. In the case of the Calculus of Inductive Constructions, the language allows us to represent all computable functions. Hence, by Turing completeness, every recursive language can be expresses as a conditioned grammar.

We will examine now two techniques for encoding conditioned grammars. The first is actually feasible in CIC; the second is a proposal for an extension of CIC. We will use the language *DiffNats* as a running example for both solutions.

11.4.2 The only solution - so far

Define a *subtype* (a *quasi-subset type*):

- represent the set of *pre-phrases* (also those not acceptable) by a type, say T_1, following the methodology described in the previous Sections.
- define a judgement $P : T_1 \to \mathit{Prop}$ on this type, inductively, which represents the condition of well-formness.
- define the type $T_2 = \Sigma_{t:T_1}(P\ t)$ of well formed phrases.

Example 11.10 Following this approach, the Coq encoding of *DiffNats* is as follows:

```
Inductive preDiffNats : Set :=
           empty : preDiffNats
          | cons : nat -> preDiffNats -> preDiffNats.

Inductive notin [n:nat] : preDiffNats -> Prop :=
          notin_empty   : (notin n empty)
        | notin_cons    : (m:nat)(l:preDiffNats)
                          ~(n=m) -> (notin n l) -> (notin n (cons m l)).

Inductive isgood    : preDiffNats -> Prop :=
          isg_empty : (isgood empty)
        | isg_cons  : (n:nat)(l:preDiffNats)
                      (notin n l) -> (isgood l) -> (isgood (cons n l)).

Record DiffNats : Set := mkdn {
                l : preDiffNats;
                c : (isgood l)
                }.
(* this is a macro for the following definition :
Inductive DiffNats : Set :=
          mkdn : (l:preDiffNats)(isgood l) -> DiffNats.
*)
```

□

This solution is not very satisfactory because

- The encoding function maps a well formed phrase in a pair $\langle t, d\rangle$, such that $t : T_1$ is the encoding of the phrase, and $d : (P\ t)$ the (encoding of the) proof of well formness. Since, usually, we do not take into account of well-formness proofs in the pre-formal syntax, adequacy theorem can be obtained only modulo the second components of these pairs. A bijection can be recovered by bundling up into the pre-formal syntax also the proofs of well-formness.
- the type T_2 is a Σ-type; hence it can be obtained only in higher order logical frameworks, such as CIC, or in Martin-Löf's type theory, but not in the Edinburgh LF.
- when we need to reason by induction on phrases, the only inductive principle we are given is that of pre-phrases, and hence we need to take into account also non well formed phrases, even if we restrict ourselves to well formed ones.

11.4.3 A proposal for extending CIC and Coq

A better solution could be available if we extended CIC with the *stratified mutual inductive definitions*. The most natural, elegant, efficient and powerful encoding of conditioned languages would be defining *simultaneously* the set of phrases and the conditions judgement. This would take the following form:

```
Mutual Inductive T1 : Set :=
```

```
        ...constructors for T1, among them there is...
        c : (A1:TA1)..(An:TAn)(isok A1 ... An) -> T1
...
with  isok : TA1 -> ... TAn -> Prop :=
        ...rules for isok... .
```

where `isok` represents the well formed condition on the constructor `c`.

Example 11.11 In the case of *DiffNats*, the encoding would appear as follows:

```
Inductive DiffNats : Set :=
          empty : DiffNats
        | cons : (n:nat)(l:DiffNats)(notin n l) -> DiffNats.
with notin : nat -> DiffNats -> Prop :=
          notin_empty : (notin n empty)
        | notin_cons  : (m:nat)(l:DiffNats)
                        ~(n=m) -> (notin n l) -> (notin n (cons m l)).
```

□

This scheme can be generalized in the notion of *stratified mutual inductive definitions*: consider the general schemata of mutual inductive definition:

```
Mutual Inductive P1 : TP1 :=
        ... constructors for P1 ...
...
with  Pk : TPk :=
        ... constructors for Pk ...
```

where `TP1...TPk` are the types of the judgements under definitions. In the actual definition of CIC, these types cannot contain the judgements under definition.

Indeed, a *tout court* elimination of this restriction yields inconsistencies. For instance, suppose that the following definition would be feasible:

```
Mutual Inductive A:B := a : B -> A
with              B:A := b : A -> B.
```

the minimal solution of the functor corresponding to this definition would be "A inhabits B which inhabits A", and this yields (a version of) the Burali-Forti paradox.

However, we do not need so general definitions. We propose to allow only for *stratified* types, that is $\mathtt{TP}_{i+1}$ can contain $\mathtt{P}_1, \ldots, \mathtt{P}_i$, and constructors of $\mathtt{P}_{i+1}$ can refer to constructors of $\mathtt{P}_1, \ldots, \mathtt{P}_i$.

As a consequence of this, we obtain:

- The definition of every conditioned grammar would fall into this category, hence a very large class of languages could be faithfully and immediately represented.
- The definition of the encoding function would be a simple extension of the usual one (provided the definition of a pre-formal proof system for the conditions)
- The inductive principles associated with these definitions would be the intuitive ones; we have not to deal with non well-formed phrases any more. For instance, in the case of *DiffNats* the inductive principle would be as follows:

```
DiffNats_ind    : (P:DiffNats->Prop)
        (P empty)
        ->((n:nat)(p:DiffNats)(notin n p) -> (P p) -> (P (cons n p)))
        ->(p:DiffNats)(P p).
```

Notice that, in the inductive step, a well-formness condition on the head natural is locally assumed, together with the inductive hypothesis. This is all we need for reasoning inductively on these lists.

- Adequacy theorems are easily adapted to this case.

Although we have not thoroughly investigated the consistency of the resulting system, no problems should arise with this extension. Peter Dybjer has introduced a similar notion in the context of Martin-Löf's type theory (*simultaneous inductive-recursive definitions*), from a categorical-theoretic viewpoint [Dyb96]. The consistency of Dybjer's extension to MLTT should be sufficient to our purposes, since in the representation of a language there is no need of impredicative features, and therefore the predicative fragment of CIC suffices.

11.4.4 Subsorting

One of the strong points of adopting the multi-sorted approach of the theory of arities, as we have seen, is that the type checking of the object language is delegated to the typing system of the metalanguage. In presence of subsorts, however, this simple approach fails, because so far Logical Frameworks based on type-theory do not support subtyping (This is an ongoing research; see the section about related works).

The direct extension of the standard approach is to add *coercion constructors* (called simply "coercions"). A coercion is just a "sort-conversion" constant for embedding expressions of a subsort in the supersort. For instance, if $S_1 \subseteq S_2$ are two sorts, represented by θ_1, θ_2 respectively, then we have to add a constant $c : \theta_1 \to \theta_2$ in the definition of θ_2.

If the Hasse diagram of the subsorting relation does not form cycles, then the use of coercions is not problematic. Otherwise, in the case there are cycles, then an expression may be represented by more than just one term. Let $S_1, S_2, S_3, S4$ be four sorts, represented respectively by $\theta_1, \ldots, \theta_4$. Suppose that $S_1 \subseteq S_i \subseteq S4$ for $i = 2, 3$; in this case, we would introduce four coercions

$$c_{12} : \theta_1 \to \theta_2 \quad c_{13} : \theta_1 \to \theta_3 \quad c_{24} : \theta_2 \to \theta_4 \quad c_{34} : \theta_3 \to \theta_4$$

In this situation, an expression $e \in S_1$ would be represented by a term $\varepsilon(e) : \theta_1$, which would be embedded in two ways into the type θ_4. In fact, both $c_{24}(c_{12}(\varepsilon(e)))$ and $c_{34}(c_{13}(\varepsilon(e)))$ have type θ_4, and correspond to the same term. In this case, therefore, we miss the bijection between expressions of sort S_4 and terms inhabiting θ_4.

Therefore, if one does not want to reason "up to coercions", a solution is to represent the language $\mathcal{L}$ by a single sort $L : \mathsf{Set}$, which represents the whole universe of expressions, with no sort structure. For each expression constructor e we introduce a term constructor εe in L, whose type is obtained by "currying" and flattening the arity of e. Each expression $e \in \mathit{Expr}$ is then mapped into a first-order term $\varepsilon(e)$ whose type is L.

The resulting encoding lacks the information carried by a multi-sorted language. For instance, in FOL, two sorted expression constructors such as $+$ and ff, whose arities are

(*Term*, *Term*) $\to$ *Term* and *Form* respectively, are represented by two constants $+'$ and ff' whose sorts are $L \to L \to L$ and L respectively. Hence, also non-well formed formulæ arise. Therefore, we need to implement "by hand" the typing system of the object language: for each sort S, we have to introduce a judgement $Wf_S : L \to \mathsf{Prop}$ on L which specifies when a term represents an expression of sort S. In this setting, a sort S is represented as a subset of the whole language L.

The use of an unique-sorted encoding is usually more cumbersome than the corresponding multisorted approach, since type checking is not delegated to the metalanguage. We do not investigate further this point; see e.g. [Ter95] for more details.

11.5 Related Work

The challenge of combining higher-order abstract syntax and inductive definitions is an ongoing and very active research area; see e.g. [DH94, DFH95, PW95, DPS96]

In [DH94], a different approach is followed. There, like in this thesis and in [DFH95], a separate type for variables is introduced. However, terms of the object language Φ are there represented as *functions* of type $LL = (list\ L) \to L$, from lists of arguments of type L to terms of type L, instead of using directly terms of type L. Despite a much more complex representation of the object-level syntax, this approach allow for short definitions of (standard) substitutions; for instance, the β-rule of λ-calculus can be encoded as follows:

$$red_\beta \quad : \quad \prod_{e,v:LL} (red\ (\mathcal{A}pp\ (\mathcal{L}am\ e)\ v)\ \lambda x{:}(list\ L).(e\ (cons\ (v\ x)\ x)))$$

This definition is just one line long, instead of the complex inductive/recursive definitions presented above and in [DFH95]. This style of presentation might be a good approach to describe a new type theory which will allow primitive recursion over higher-order abstract syntax.

Other approaches aim to extend the underlying metalanguage, so that it can be used for describing *primitive recursive* functional types. Since primitive recursion cannot be allowed on any term of functional (i.e., higher-order) type, the problem is how to circumsribe the functional types on which primitive recursion can be allowed without giving rise to paradoxes. In [DPS96], an approach inspired by Linear Logic is followed: the primitive recursive function space $A \Rightarrow B$ is decomposed into a "modal type operator" and a parametric function space: $A \Rightarrow B = (\Box A) \to B$. This promising approach has been proved successful in the case of simple typed λ-calculus; ongoing researches (e.g. by Despeyroux and Leleu at INRIA-Sophia Antipolis) aim to extended it approach to λ-calculi with dependent types.

Also the issue of combining subsorting with dependent types is an ongoing research area; we refer the reader to [Com94, KP93b, Luo96].

Chapter 12

Encoding of Operational Semantics

From a logician's point of view, a Structural or Natural Operational Semantics of a language is just a formal logical system. In particular, a NOS specification is a formal system in Natural Deduction style, therefore it can be easily encoded in interactive proof checkers based on type-checking of typed λ-calculus, such as those presented in Chapter 10. Moreover, as we have already pointed out in Chapter 4, a Natural Deduction-style presentation of the semantics is more modular. Furthermore, Natural Deduction style evaluation proofs are compositional: any proof can be reused in more complex proofs, provided that the assumptions on which it depends are preserved.

In this chapter we study the encoding of SOS and NOS specifications in Logical Frameworks. As for any other formal system, in encoding the semantics we aim at:

- *a faithful representation of the system*: an encoding function from the pre-formal proofs and the formal syntax of terms of a certain type. *Side* and *proof conditions* on derivations have to be enforced;
- *taking full advantage* of the features of a Logical Framework (higher-order, multi-judgement), for reducing the overhead in encoding and using the systems;
- *a way for reasoning on it*, namely by induction. Most proofs are done by structural induction on the length of computation [Plo81].

The chapter is organized as follows. In Section 12.1 we introduce a general methodology for encoding transition systems. Despite its generality, the outlined methodology is not very satisfactory if we are dealing with stores and/or environments; therefore, in Section 12.2 we focus on the encoding of specifications in Natural Operational Semantics. Finally, in Section 12.3 we provide some references to other approaches not faced here.

12.1 The basic approach

Natural Semantics and SOS are defined by sets of rules with *no* discharged assumptions. These rules a set of evaluation judgements which possibly involve linear data structures such as environment and stores (see Section 4.1). A general implementation is to adopt straightforwardly the LF methodology of [HHP93]:

- each evaluation judgement is represented by an inductively defined judgement
- each inference rule is represented by a constructor.

Example 12.1 Let us consider the following TAP, **t**rivial **a**lgebra of **p**rocesses, whose syntax and operational semantics are defined as follows:

$$\boxed{\alpha ::= a \mid b \mid ct ::= nil \mid \alpha.t \mid t + t}$$

$$\boxed{\alpha.t \to t \quad \frac{t_1 \to t}{t_1 + t_2 \to t} \quad \frac{t_2 \to t}{t_1 + t_2 \to t}}$$

Let us denote by $\pi :\vdash t_1 \to t_2$ a derivation of $t_1 \to t_2$ in such system. This specification can be directly represented in CIC by the following signature Σ (in Coq syntax):

```
Inductive Set Act := a:Act | b:Act | c:Act.

Inductive Set Proc :=
             nil  : Proc
           | prfx : Act -> Proc -> Proc
           | or   : Proc -> Proc -> Proc.

Inductive trans  : Proc -> Proc -> Prop :=
          action : (ac:Act)(t:Proc)(trans (prfx ac t) t)
        | ndc1   : (t1,t2,t:Proc)(trans t1 t) -> (trans (or t1 t2) t)
        | ndc2   : (t1,t2,t:Proc)(trans t2 t) -> (trans (or t1 t2) t).
```

The encoding functions:

$$\varepsilon_0(a) = \texttt{a} \qquad \varepsilon_0(b) = \texttt{b} \qquad \varepsilon_0(c) = \texttt{c}$$
$$\varepsilon_1(nil) = \texttt{nil} \qquad \varepsilon_1(\alpha.t) = (\texttt{prfx}\ \varepsilon_0(\alpha)\ \varepsilon_1(t)) \qquad \varepsilon_1(t_1 + t_2) = (\texttt{or}\ \varepsilon_1(t_1)\ \varepsilon_1(t_2))$$

$$\begin{aligned}
\varepsilon_2(\alpha.t \to t) &= (\texttt{action}\ \varepsilon_0(\alpha)\ \varepsilon_1(t)) \\
\varepsilon_2\left(\frac{\pi}{\dfrac{t_1 \to t}{t_1 + t_2 \to t}}\right) &= (\texttt{ndc1}\ \varepsilon_1(t_1)\ \varepsilon_1(t_2)\ \varepsilon_1(t)\ \varepsilon_2(\pi)) \\
\varepsilon_2\left(\frac{\pi}{\dfrac{t_2 \to t}{t_1 + t_2 \to t}}\right) &= (\texttt{ndc2}\ \varepsilon_1(t_1)\ \varepsilon_1(t_2)\ \varepsilon_1(t)\ \varepsilon_2(\pi))
\end{aligned}$$

It is easy to prove that these encoding functions are bijective:

Proposition 12.1 (Adequacy for TAP) *The map ε_0 is a bijection between Act and canonical terms a such that $\vdash_\Sigma a : Act$; the map ε_1 is a bijection between Proc and canonical terms t such that $\vdash_\Sigma t : Proc$.*

For all $t_1, t_2 \in$ Terms, ε_2 is a bijection between proofs in the SOS of $t_1 \to t_2$, and canonical terms d such that $\vdash_\Sigma d : (\texttt{trans}\ \varepsilon_1(t_1)\varepsilon_2(t_2))$.

□

In many cases, this solution is satisfactory:

- easily deduced by the SOS (or Natural Semantics) specification; actually, it can be mechanized;
- adequacy theorem is easy;
- induction principles on proofs (=computations) are available.

The application of this approach to the Natural Semantics paradigm has been studied in depth by Terrasse [Ter95], (although with a different treatment of syntax). As Terrasse pointed out, the process of translation from a high-level formalism for Natural Semantics (Typol) to this kind of encoding is mechanizable.

However, this approach suites very well only "propositional semantics," that is operational semantics with neither environments nor stores, nor processes variables. As we have seen in Section 4.1.2, bundling environments and stores into the judgements has some drawbacks; moreover, this approach does not take advantage of higher-order features of logical framework.

For these reasons, we focus on the *Natural Operational Semantics* paradigm, which we have describe in Section 4.2.

12.2 Encoding of Natural Operational Semantics

Due to their strong Natural Deduction flavour, NOS specifications as described in Section 4.2 are directly encoded in logical frameworks and implemented within LF-based proof environments, giving us powerful tools for developing language semantics formally, for checking correctness of translators and for proving semantic properties. See for instance [MP91, Mic94].

The encoding of NOS has several significant consequences. In fact, when we encode the operational semantics, we have to discuss details that are normally left out, or taken for granted. Most of the rules of a NOS specification are directly encoded without any particular difficulty. However, particular care has to be taken in implementing the core idea of Natural Operational Semantics, that is the distribution of informations into the derivation context by means of bookkeepings (see Section 4.2.2). For the sake of simplicity, we focus on the following **let** rule; other "substitution-like" rules (such as rule A.1) are encoded similarly.

$$\frac{\begin{array}{c}(x' \Rightarrow n)\\ \vdots \\ N \Rightarrow n \quad M' \Rightarrow m'\end{array}}{\textbf{let } x = N \textbf{ in } M \Rightarrow m}\ \mathsf{EC}(x, M, m)$$

where the eigenvariable side condition $\mathsf{EC}(x, M, m)$ is

$$\mathsf{EC}(x, M, m) \quad \equiv \quad \begin{array}{l}\text{"}M', m' \text{ are obtained from } M, m \text{ respectively by replacing } \textit{all} \text{ the}\\ \text{occurrences of } x \text{ with } x', \text{ which does not appears neither in } x,\ M,\\ m \text{ nor in any live assumption different from } (x' \Rightarrow n)\text{"}.\end{array}$$

The problem is to enforce precisely this side condition. The canonical, paradigmatical approach to similar capture-avoiding substitution problems is based on the idea of *higher-order abstract syntax* (Chapter 2 and Section 10.5): every binding operator is reduced to the λ-abstraction of the metalanguage [HHP93, Gar92]. Accordingly to this idea, variables of the object language are represented by variables of the metalanguage. In this way, we immediately recover both α-conversion and capture-avoiding substitution at the object language level, from the α-conversion and the β-reduction of metalanguage, respectively.

Recall that, as we have pointed out in Section 11.1, the feasibility of HOAS is submitted to precise closure properties of both the syntax and the proof system. Roughly, let S be a sort represented by HOAS by a type θ_S, and let x a variable ranging over S (and hence represented by a metavariable $x : \theta_S$); then, every occurrence of x in any phrase φ and proof π can be replaced by any term $t \in S$, leading to the existence of $\varphi[t/x]$ and $\pi[t/x]$ (see Theorem 11.2 and Corollary 11.3 for precise statements). Now, these "closure properties" often are unsound in presence of commands and other imperative figure (see also Section 3.2.3): actually, a variables can be seen as a placeholder for terms only in purely functional languages.

In the case we cannot adopt HOAS directly, we introduce a specific set for variables,

```
Parameter Var:Set.
```

which is embedded into terms by a suitable *coercion constructor*:

```
Mutual Inductive
     Expr : Set
        := isX   : Var -> Expr
        |  zero  : Expr
        |  succ  : Expr -> Te
        |  do    : Prog -> Expr -> Expr
with Prog : Set
        := ass   : Var -> Expr -> Prog
        |  comp  : Prog -> Prog -> Prog
        |  while : Expr -> Prog -> Prog.
```

For implementing the bookkeepings, we introduce the auxiliary judgement $\mapsto$: $Var \to Expr \to \mathsf{Prop}$. Therefore, the evaluation of identifiers is realized by an application of the constant

$$\textsc{Eval\_Var} : \prod_{x:Var} \prod_{M:Expr} (x \mapsto M) \to ((IsX\ x) \Rightarrow M)$$

The most difficult task is to express formally the "freshness" condition of substitution rules. We need to reify metasyntactic notions such as "occurrence" and "non-occurrence" of variables in terms. For the sake of simplicity, we focus on the language of **while** programs. This is achieved by introducing two auxiliary judgements:

$$isin, isnotin : Var \to \prod_{S:\mathsf{Set}} S \to \mathsf{Prop}.$$

The intended meaning of "*isin* x S t" is "the variable x appears in the phrase t which belongs to the syntactic class S"; dually for *notin*. The rules for these judgements are

given on the syntax of phrases, accordingly to the intuitive notion of "occurrence" and "nonoccurrence", similarly to those of Dynamic Logic (Figure 14.5).

Thus, the complete substitution rule for **let** is as follows (we present it in a pretty-printed Natural Deduction-style fashion)

$$M, m{:}\,Var \to Expr \quad \frac{N \Rightarrow n \qquad \forall w{:}var \begin{cases} (isnotin\ x\ \ Var\ w) \\ \vdots \\ isnotin\ x\ \ Expr\ (M\ w) \end{cases} \qquad \forall x'{:}\,Var \begin{cases} \begin{pmatrix} (isnotin\ x'\ \ Var\ x) \\ (isnotin\ x'\ \ Expr\ (M\ x)) \\ (isnotin\ x'\ \ Expr\ (m\ x)) \\ x' \mapsto n \end{pmatrix} \\ \vdots \\ (M\ x') \Rightarrow (m\ x') \end{cases}}{\mathbf{let}\ x = N\ \mathbf{in}\ (M\ x) \Rightarrow (m\ x)}$$

The middle subderivation requires $M(\cdot)$ is a "good" context for x, that is, x does not occur in the expression context M (expression with a hole). Evaluation is performed in the right-hand subderivation; here, x is replaced by x' assuming x' does not occur in any of $x, (M\ x), (m\ x)$. This is achieved by the three discharged assumptions about *isnotin*; these assumptions can be needed in the subderivation in order to prove other well-formness conditions on contexts, such as the one on M.

12.3 Related work

The encoding of various forms of natural semantics in Logical Frameworks has been subject to an extensive study. In [Ter95], Terrasse has investigated the automatic translation of TYPOL specifications into Coq signatures.

An interesting improvement of Natural Semantics that we have not investigated here, is the use of higher-order abstract syntax in representing binding operators. This approach, introduced by Hannan in the *Extended Natural Semantics* (ENS) [Han93], has been proved particularly suited for the specification of purely functional languages. Although in Hannan's original approach, ENS specifications are encoded in λProlog, the higher-order solution has been successfully applied in other Logical Frameworks. In the Elf logic programming language (which is based on the Edinburgh LF), ENS has been used by Michaylov and Pfenning for investigating the meta-theoretic properties of operational semantics [MP91], and by Harper and Pfenning for addressing the issue of compiler correctness [HP92]. In the Forum language (which is based on Linear Logic), higher-order Natural Semantics specifications have been used by Miller [Mil94] and Chirimar [Chi95].

However, higher-order abstract syntax cannot be applied in presence of imperative features (as we have see in Section 4.2). The NOS paradigm aims to overcome this problem; in this Chapter we have addressed the issue of encoding NOS specifications in a type-theory based Logical Framework.

Chapter 13

Encoding of Modal Logics

In this chapter we examine some basic techniques for representing modal features of proof systems. These techniques are useful not only in encoding truly modal logics, such as $\mathbf{N}K$, but any proof system with "modal rules", that is, rules in which we have to check dependencies of conclusion from assumptions.

13.1 Encoding of the syntax

The encoding of the syntax of modal formulæ is not problematic; it follows the standard pattern of free algebræ. For the sake of simplicity, but without loss of generality, we focus on a minimal fragment, namely

$$\Phi: \quad \varphi ::= p \mid \varphi \supset \psi \mid \Box\varphi$$

which is encoded by the following signature:

$o : \mathsf{Set}$	`Inductive o : Set :=`
$p : o$	`p : o`
$\supset : o \to o \to o$	`| imp : o-> o-> o`
$\Box : o \to o$	`| box : o-> o`

In this simple case, we can apply freely the *variable convention*: we do not need of introducing a specific type for propositional variables; instead, we represent propositional variables by means of metalinguistic variables of type o. The easy encoding function is then

$$\varepsilon_X : \Phi_X \to \{t \mid \Xi(X) \vdash t : o, t \text{ canonical}\}$$

defined as usual. Let δ_X be the left-inverse of ε_X.

In order to simplify the presentation, in the rest of the chapter we will sometime drop the encoding; we will hence denote by "φ" both the abstract formula $\varphi \in \Phi$ and its encoding $\varepsilon_X(\varphi) : o$.

13.2 Encoding of systems for K

Here we give two methodologies for representing modalities and proof rules for $\mathbf{N}K$. Let us recall the $\Box'$-I rule:

$$\frac{\emptyset \vdash \varphi}{\emptyset \vdash \Box\varphi} \quad \text{that is,} \quad \frac{\Gamma \vdash \varphi \;\; \varphi \text{ does not depend on any assumption in } \Gamma}{\Gamma \vdash \Box\varphi}$$

U	:	Set
T	:	$U \to o \to \mathsf{Prop}$
$\supset$-I	:	$\prod_{\varphi,\psi:o} \prod_{w:U} ((T\ w\ \varphi) \to (T\ w\ \psi)) \to (T\ w\ (\supset\ \varphi\psi))$
$\supset$-E	:	$\prod_{\varphi,\psi:o} \prod_{w:U} (T\ w\ (\supset\ \varphi\psi)) \to (T\ w\ \varphi) \to (T\ w\ \psi)$
$\supset_\Box$-E	:	$\prod_{\varphi,\psi:o} \prod_{w:U} (T\ w\ \Box(\supset\ \varphi\psi)) \to (T\ w\ \Box\varphi) \to (T\ w\ \Box\psi)$
$\Box'$-I	:	$\prod_{\varphi:o} (\prod_{w:U} (T\ w\ \varphi)) \to \prod_{w:U} (T\ w\ (\Box\varphi))$

$\Box$-E	:	$\prod_{\varphi:o} \prod_{w:U} (T\ w\ \Box\varphi) \to (T\ w\ \varphi)$
$\Box_\Box$-I	:	$\prod_{\varphi:o} \prod_{w:U} (T\ w\ \Box\varphi) \to (T\ w\ \Box\Box\varphi)$
$\Box_\Diamond$-I	:	$\prod_{\varphi:o} \prod_{w:U} (T\ w\ \Diamond\varphi) \to (T\ w\ (\Box\Diamond\varphi))$
$\Box_\supset$-E	:	$\prod_{\varphi:o} \prod_{w:U} (T\ w\ \Box(\supset\ (\Box\varphi)\varphi)) \to (T\ w\ \Box\varphi)$

Figure 13.1: $\Sigma_w(\mathbf{N}K)$ and its extensions for $\mathbf{N}K4$,...

The two methodologies we are going to describe correspond to two dual approaches. Roughly, in the first one, called "world technique", we make explicit the dependencies of the conclusion from the assumptions by "tagging" the proving judgements. On the other hand, in the second technique we do not change the proving judgement; instead, we decide what are the assumptions a proof depends on by examining its representing proof term. We will see that these two approaches are strictly related.

13.2.1 World Parameter

In Figure 13.1 we give the signature $\Sigma_w(\mathbf{N}K)$ and its extensions for the other systems ($\mathbf{N}K4$, ...).

The extra sort U (the *universe*) has no constructors: therefore, the only terms inhabiting U are variables, which have to be assumed in the typing context. These variables are called suggestively "worlds" (of the universe). An assertion of the form $(T\ w\ \varphi)$ has the intuitive meaning "φ holds in the world w." It should be noticed, however, that these names are chosen only for their intuitive meaning, and there is no direct connection with Kripke semantics of modal logics. The idea behind the use of the extra parameter is that in making an assumption, we are forced to assume the existence of a world, say w, and to instantiate the truth judgement T also on w. This judgement appears also as an hypothesis on w. Hence, deriving as premise a judgement, which is universally quantified with respect to U, amounts to estabilishing the judgement for a generic world on which no assumptions are made, i.e., on no assumptions.

We prove formally the adequacy of this approach.

Definition 13.1 (Encoding of assumptions and proofs) *Let w be a variable of sort U, X set of propositional variables. For each $\Delta \subseteq \Phi_X$, we define the context $\gamma_w(\Delta)$ as follows:*

$$\gamma_w(\Delta) \stackrel{\text{def}}{=} \begin{cases} w : U & \text{if } \Delta \equiv \emptyset \\ \gamma_w(\Delta'), v_\varphi : (T\ w\ \varepsilon_X(\varphi)) & \text{if } \Delta \equiv \Delta', \varphi \text{ and } v_\varphi \text{ fresh for } \gamma_w(\Delta') \end{cases}$$

The encoding function for proofs,

$$\varepsilon^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w} : \{\pi \mid \pi : \Delta \vdash \varphi, \mathrm{FV}(\pi) \subseteq X\} \to \{t \mid \Xi(X), \gamma_w\Delta \vdash^{\Sigma_w(\mathbf{N}K)} t : (T\ w\ \varepsilon_X(\varphi))\}$$

is defined on the structure of proofs of $\mathbf{N}K$*; given a proof* $\pi : \Delta \vdash_{\mathbf{N}K} \varphi$*,* $\varepsilon_{X,\Delta,w}(\pi)$ *is the proof term corresponding to* π*, as follows:*

$$\begin{aligned}
\varepsilon_{X,\Delta,w}(\varphi) &\stackrel{\text{def}}{=} v_\varphi\ ,\quad if\ v_\varphi \in \gamma_w(\Delta)\\
\varepsilon_{X,\Delta,w}(\Box'\text{-I}_\varphi(\pi')) &\stackrel{\text{def}}{=} (\Box'\text{-I}\ \varepsilon_X(\varphi)\ (\lambda w' : U.\varepsilon_{X,\emptyset,w'}(\pi'))\ w)\\
\varepsilon_{X,\Delta,w}(\supset\text{-I}_{\varphi\psi}(\pi')) &\stackrel{\text{def}}{=} (\supset\text{-I}\ \varepsilon_X(\varphi)\ \varepsilon_X(\psi)\ w\ (\lambda v_\varphi{:}(T\ w\ \varepsilon_X(\varphi)).\varepsilon_{X,(\Delta,\varphi),w}(\pi')))\\
\varepsilon_{X,\Delta,w}(\supset\text{-E}_{\varphi\psi}(\pi',\pi'')) &\stackrel{\text{def}}{=} (\supset\text{-E}\ \varepsilon_X(\varphi)\ \varepsilon_X(\psi)\ w\ \varepsilon_{X,\Delta,w}(\pi')\ \varepsilon_{X,\Delta,w}(\pi''))\\
\varepsilon_{X,\Delta,w}(\supset_\Box\text{-E}_{\varphi\psi}(\pi',\pi'')) &\stackrel{\text{def}}{=} (\supset_\Box\text{-E}\ \varepsilon_X(\varphi)\ \varepsilon_X(\psi)\ w\ \varepsilon_{X,\Delta,w}(\pi')\ \varepsilon_{X,\Delta,w}(\pi''))
\end{aligned}$$

Theorem 13.1 *The function* $\varepsilon^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w}$ *is a compositional bijection between proofs* $\pi : \Delta \vdash \varphi$*, such that* $\mathrm{FV}(\pi) \subseteq X$*, and canonical terms* t*, such that* $\Xi(X), \gamma_w(\Delta) \vdash_{\Sigma_w(\mathbf{N}K)} t : (T\ w\ \varepsilon_X(\varphi))$*.*

Proof. We verify by induction on the structure of proofs that $\varepsilon^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w}(\pi)$ is a canonical term of type $(T\ w\ \varepsilon_X(\varphi))$ in $\Sigma_w(\mathbf{N}K)$ and $\Xi(X), \gamma_w(\Delta)$.
Base Step. Let $\varphi \in \Delta$ is an assumption. Since $\varepsilon^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w}(\varphi) = v_\varphi \in \gamma_w(\Delta)$, immediately $\Xi(X), \gamma_w(\Delta) \vdash_{\Sigma_w(\mathbf{N}K)} v_\varphi : (T\ w\ \varepsilon_X(\varphi))$.
Inductive Step. By cases on the last rule applied.

If $\pi \equiv \supset\text{-E}_{\psi,\varphi}(\pi',\pi'')$, then $\pi' : \Delta \vdash \psi \supset \varphi$ and $\pi'' : \Delta \vdash \psi$ with free variables in X. By IH, we have

$$\begin{aligned}
\Xi(X), \gamma_w(\Delta) &\vdash_{\Sigma_w(\mathbf{N}K)} \varepsilon^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w}(\pi') : (T\ w\ \varepsilon_X(\psi))\\
\Xi(X), \gamma_w(\Delta) &\vdash_{\Sigma_w(\mathbf{N}K)} \varepsilon^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w}(\pi'') : (T\ w\ \varepsilon_X(\psi \supset \varphi)).
\end{aligned}$$

Therefore, we have immediately,

$$\Xi(X), \gamma_w(\Delta) \vdash_{\Sigma_w(\mathbf{N}K)} \left(MP\ \varepsilon_X(\psi)\ \varepsilon_X(\varphi)\ w\ \varepsilon^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w}(\pi')\ \varepsilon^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w}(\pi'')\right) : (T\ w\ \varepsilon_X(\varphi)).$$

The case for $\supset_\Box$-E is similar.

If $\pi \equiv \supset\text{-I}_{\varphi,\psi}(\pi')$, then $\pi' : \Delta, \varphi \vdash_{\mathbf{N}K} \psi$. By IH, we have

$$\Xi(X), \gamma_w(\Delta, \varphi) \vdash_{\Sigma_w(\mathbf{N}K)} \varepsilon^{\Sigma_w(\mathbf{N}K)}_{X,(\Delta,\varphi),w}(\pi') : (T\ w\ \varepsilon_X(\psi))$$

and abstracting on v_φ we obtain

$$\Xi(X), \gamma_w(\Delta) \vdash_{\Sigma_w(\mathbf{N}K)} (\lambda v_\varphi{:}(T\ w\ \varepsilon_X(\varphi)).\varepsilon^{\Sigma_w(\mathbf{N}K)}_{X,(\Delta,\varphi),w}(\pi')) : \prod_{v_\varphi:(T\ w\ \varepsilon_X(\varphi))} (T\ w\ \varepsilon_X(\psi)).$$

Finally, applying the constant $\supset$-I we obtain

$$\begin{aligned}
\Xi(X), \gamma_w(\Delta) \vdash_{\Sigma_w(\mathbf{N}K)} &(\supset\text{-I}\ \varepsilon_X(\varphi)\ \varepsilon_X(\psi)\ w\ (\lambda v_\varphi : (T\ w\ \varepsilon_X(\varphi)).\varepsilon^{\Sigma_w(\mathbf{N}K)}_{X,(\Delta,\varphi),w}(\pi')))\\
&: (T\ w\ \varepsilon_X(\supset\ \varphi\ \psi)).
\end{aligned}$$

Otherwise, $\pi \equiv \Box'\text{-I}_\varphi(\pi')$; then $\pi' : \emptyset \vdash_{\mathbf{N}K} \varphi$. By IH, we have

$$\Xi(X), \gamma_w(\emptyset) \vdash_{\Sigma_w(\mathbf{N}K)} \varepsilon^{\Sigma_w(\mathbf{N}K)}_{X,\emptyset,w}(\pi') : (T\ w\ \varepsilon_X(\varphi)).$$

By abstracting on w we have

$$\Xi(X) \vdash_{\Sigma_w(\mathbf{N}K)} \left(\lambda w'{:}U.\varepsilon^{\Sigma_w(\mathbf{N}K)}_{X,\emptyset,w'}(\pi')\right) : \prod_{w':U} (T\ w'\ \varepsilon_X(\varphi))$$

therefore, we have immediately

$$\Xi(X), \gamma_w(\emptyset) \vdash_{\Sigma_w(\mathbf{N}K)} \left(\Box'\text{-I}\ \varepsilon_X(\varphi)(\lambda w'{:}U.\varepsilon^{\Sigma_w(\mathbf{N}K)}_{X,\emptyset,w'}(\pi'))\right) w : (T\ w\ \Box\varepsilon_X(\varphi)).$$

By the above steps, it is easy to show that $\varepsilon^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w}$ is injective. Surjectivity is established by exhibiting a left-inverse $\delta^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w}$, defined by induction on the structure of the canonical forms as follows:

$$\begin{array}{rcll}
\delta^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w}(v_\varphi) & \stackrel{\text{def}}{=} & \varphi & ,if\ v_\varphi \in \mathrm{dom}(\gamma_w(\Delta)) \\
\delta^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w}(\supset\text{-I}\ t\ t'\ w(\lambda p : (T\ w\ t).p')) & \stackrel{\text{def}}{=} & \supset\text{-I}_{\delta_X(t),\delta_X(t')}\left(\delta^{\Sigma_w(\mathbf{N}K)}_{X,(\Delta,\delta_X(t)),w}(p')\right) & \\
\delta^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w}(\supset_\Box\text{-E}\ t\ t'\ w\ p\ p') & \stackrel{\text{def}}{=} & \supset_\Box\text{-E}_{\delta_X(t),\delta_X(t')}\left(\delta^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w}(p), \delta^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w}(p')\right) & \\
\delta^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w}(\supset\text{-E}\ t\ t'\ w\ p\ p') & \stackrel{\text{def}}{=} & \supset\text{-E}_{\delta_X(t),\delta_X(t')}\left(\delta^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w}(p), \delta^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w}(p')\right) & \\
\delta^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w}(\Box'\text{-I}\ t\ (\lambda w'{:}U.p)\ w) & \stackrel{\text{def}}{=} & \Box'\text{-I}_{\delta_X(t)}\left(\delta^{\Sigma_w(\mathbf{N}K)}_{X,\emptyset,w}(p)\right) &
\end{array}$$

The decoding map $\delta^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w}$ is total and well-defined. Moreover, if $\Xi(X), \gamma_w(\Delta) \vdash_{\Sigma_w(()} \mathbf{N}K)p : (T\ w\ t)$ then $\delta^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w}$ is a proof of $\Delta \vdash \delta_X(t)$; we show this by induction on the syntax of canonical form p of type $(T\ w\ t)$.

Base Step. Let be $p \equiv v_\varphi : (T\ w\ \varepsilon_X(\varphi))$; then, take $\pi = \varphi$.

Inductive Step. We see only three significant cases.

If $p \equiv (\supset\text{-E}\ t'\ t''\ w\ p'\ p'')$:$(T\ w\ t'')$, since p is well-typed, we have

$$\begin{array}{lll}
\Xi(X), \gamma_w(\Delta) & \vdash_{\Sigma_w(\mathbf{N}K)} & p' : (T\ w\ t') \\
\Xi(X), \gamma_w(\Delta) & \vdash_{\Sigma_w(\mathbf{N}K)} & p'' : (T\ w\ (\supset\ t'\ t'')).
\end{array}$$

By IH, there are two proofs $\delta^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w}(p') : \Delta \vdash_{\mathbf{N}K} \delta_X(t')$ and $\delta^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w}(p'') : \Delta \vdash_{\mathbf{N}K} \delta_X(\supset t'\ t'')$. Therefore by applying $\supset$-E we obtain the proof $\pi : \Delta \vdash_{\mathbf{N}K} \delta_X(t'')$.

If $p \equiv (\supset\text{-I}\ t'\ t''\ w\ p') : (T\ w\ (\supset\ t'\ t''))$, since p is well-typed, we have $p' \equiv \lambda v{:}(T\ w\ t').p''$ and

$$\Xi(X), \gamma_w(\Delta) \vdash_{\Sigma_w(\mathbf{N}K)} p' : (T\ w\ t') \to (T\ w\ t'')$$

that is,

$$\Xi(X), \gamma_w(\Delta), v{:}(T\ w\ t') \vdash_{\Sigma_w(\mathbf{N}K)} p'' : (T\ w\ t'')$$

Let $\varphi \stackrel{\text{def}}{=} \delta_X(t')$; then, $\gamma_w(\Delta), v{:}(T\ w\ t') = \gamma_w(\Delta, \varphi)$. By IH, there is a proof $\delta^{\Sigma_w(\mathbf{N}K)}_{X,(\Delta,\varphi),w}(p') : \Delta, \varphi \vdash_{\mathbf{N}K} \delta_X(t'')$. Therefore by applying $\supset$-I we obtain the proof $\pi : \Delta \vdash_{\mathbf{N}K} \varphi \supset \delta_X(t'')$.

If $p \equiv (\Box'\text{-I}\ t'\ (\lambda w'{:}U.p')w) : (T\ w\ (\Box t'))$, since p is well-typed, we have that

$$\Xi(X), \gamma_w(\Delta) \vdash_{\Sigma_w(\mathbf{N}K)} (\lambda w'{:}U.p') : \prod_{w':U} (Tw't').$$

Notice that each (canonical) term p of type $(T\ w\ t)$ has exactly one free variable of type U, namely w. This can be proved by induction on the structure of p (look at the previous steps). Hence, $(\lambda w'{:}U.p')$ has no free variable of type U. We can drop therefore the hypotheses $\gamma_w(\Delta)$, since if they appear free in p there should be two free variables of type U in p' — a contradiction. Hence,

$$\Xi(X) \vdash_{\Sigma_w(\mathbf{N}K)} (\lambda w'{:}U.p'){:} \prod_{w':U} (T\ w'\ t')$$

that is

$$\Xi(X), w'{:}U \vdash_{\Sigma_w(\mathbf{N}K)} p'{:}(T\ w'\ t').$$

By IH there is a valid proof $\delta^{\Sigma_w(\mathbf{N}K)}_{X,\emptyset,w'}(p') : \emptyset \vdash_{\mathbf{N}K} \delta_X(t')$. Hence by applying $\Box$-I we obtain the proof $\pi : \Delta \vdash_{\mathbf{N}K} \delta_X(\Box t')$.

It remains to show that $\delta^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w}\left(\varepsilon^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w}(\pi)\right) = \pi$, and that $\varepsilon^{\Sigma_w(\mathbf{N}K)}_{X,\Delta,w}$ is compositional. This is proved by induction on the structure of π, following the steps above. □

13.2.2 Closed Judgement

In Figure 13.2 we give the signature $\Sigma_{Cl}(\mathbf{N}K)$ and its extensions for the other truth systems ($\mathbf{N}K4$, $\mathbf{N}KT$, ...). Notice that there is a rule for establishing the "closed assumption"-judgement corresponding to each proof constructor, i.e. for each rule in $\mathbf{N}K$.

The existence and definition of the encoding function relies upon two technical lemmata:

Lemma 13.2 *$\forall p$ canonical form,*
if $\Xi(X), \Delta \vdash_{\Sigma_{Cl}(\mathbf{N}K)} p : (T\ t)$ *then* $\exists c.\Xi(X), \Delta, \Xi_p(\Delta) \vdash_{\Sigma_{Cl}(\mathbf{N}K)} c : (Cl\ t\ p)$,
where $\Xi_p(\Delta) \stackrel{\text{def}}{=} \{c{:}(Cl\ t\ x) \mid x \in \mathbb{FV}(p) \wedge (x{:}(T\ t)) \in \Delta\}$.

Proof. (Hint) Long and tedious induction on p □

Lemma 13.2 defines naturally a function from canonical proof forms $p : (T\ t)$ to canonical forms of type $(Cl\ t\ p)$, in the same environment expanded with the "closed assumptions" for the free variables of p. Let us denote such function by α.

Lemma 13.3 *$\forall c$ canonical form,*
if $\Gamma_X, \Delta, \Xi \vdash_{\Sigma_{Cl}(\mathbf{N}K)} c{:}(Cl\ t\ p)$ *then* $\Gamma_X, \Delta', \Xi \vdash_{\Sigma_{Cl}(\mathbf{N}K)} c{:}(Cl\ t\ p)$,
where Ξ *contains all and only the Cl assertions, and* $\Delta' = \{x{:}(T\ t) \mid (Cl\ t\ x) \in \Im(\Xi)\}$.

We can now define the encoding function $\varepsilon^{\Sigma_{Cl}(\mathbf{N}K)}_{X,\Delta}$, which relies on the α abovementioned.

$$\begin{array}{l}
\varepsilon^{\Sigma_{Cl}(\mathbf{N}K)}_{X,\Delta}(\varphi) \stackrel{\text{def}}{=} v_\varphi \qquad\qquad ,if\ \varphi \in \Delta \\
\varepsilon^{\Sigma_{Cl}(\mathbf{N}K)}_{X,\Delta}(\supset\text{-I}_{\varphi,\psi}(\pi)) \stackrel{\text{def}}{=} \supset\text{-I}\ \varepsilon_X(\varphi)\ \varepsilon_X(\psi)\ (\lambda v_\varphi{:}(T\ \varepsilon_X(\varphi)).\varepsilon^{\Sigma_{Cl}(\mathbf{N}K)}_{X,(\Delta,\varphi)}(\pi)) \\
\varepsilon^{\Sigma_{Cl}(\mathbf{N}K)}_{X,\Delta}(\supset\text{-E}_{\varphi,\psi}(\pi',\pi'')) \stackrel{\text{def}}{=} \supset\text{-E}\ \varepsilon_X(\varphi)\ \varepsilon_X(\psi)\ \varepsilon^{\Sigma_{Cl}(\mathbf{N}K)}_{X,\Delta}(\pi')\ \varepsilon^{\Sigma_{Cl}(\mathbf{N}K)}_{X,\Delta}(\pi'') \\
\varepsilon^{\Sigma_{Cl}(\mathbf{N}K)}_{X,\Delta}(\supset_\Box\text{-E}_{\varphi,\psi}(\pi',\pi'')) \stackrel{\text{def}}{=} \supset_\Box\text{-E}\ \varepsilon_X(\varphi)\ \varepsilon_X(\psi)\ \varepsilon^{\Sigma_{Cl}(\mathbf{N}K)}_{X,\Delta}(\pi')\ \varepsilon^{\Sigma_{Cl}(\mathbf{N}K)}_{X,\Delta}(\pi'') \\
\varepsilon^{\Sigma_{Cl}(\mathbf{N}K)}_{X,\Delta}(\Box'\text{-I}_\varphi(\pi)) \stackrel{\text{def}}{=} \Box'\text{-I}\ \varepsilon_X(\varphi)\ \varepsilon^{\Sigma_{Cl}(\mathbf{N}K)}_{X,\emptyset}(\pi)\ \alpha\left(\varepsilon^{\Sigma_{Cl}(\mathbf{N}K)}_{X,\emptyset}(\pi)\right)
\end{array}$$

Judgements

$$T : lo \to \mathsf{Prop}$$
$$Cl : l\prod_{\varphi:o}(T\ \varphi) \to \mathsf{Prop}$$

Axioms and Rules

$$\supset_\Box\text{-E} : l\prod_{\varphi,\psi:o}(T\ \Box(\supset\varphi\psi)) \to (T\ \Box\varphi) \to (T\ \Box\psi)$$
$$\supset\text{-I} : l\prod_{\varphi,\psi:o}((T\ \varphi) \to (T\ \psi)) \to (T\ (\supset\varphi\psi)),$$
$$\supset\text{-E} : l\prod_{\varphi,\psi:o}(T\ (\supset\varphi\psi)) \to (T\ \varphi) \to (T\ \psi),$$
$$\Box'\text{-I} : l\prod_{\varphi:o}\prod_{d:(T\ \varphi)}(Cl\ \varphi\ d) \to (T\ \Box\varphi),$$
$$Cl_{\Box'\text{-I}} : l\prod_{\varphi:o}\prod_{d_1:(T\ \varphi)}\prod_{c_1:(Cl\ \varphi\ d_1)}(Cl\ \Box\varphi\ (\Box'\text{-I}\ \varphi\ d_1\ c_1))$$
$$Cl_{\supset\text{-I}} : l\prod_{\varphi,\psi:o}\prod_{d:(T\ \varphi)\to(T\ \psi)}\left(\prod_{x:(T\ \varphi)}(Cl\ \varphi\ x) \to (Cl\ \psi\ dx)\right) \to (Cl\ \psi\ (\supset\text{-I}\ \varphi\ \psi\ d)),$$
$$Cl_{\supset\text{-E}} : l\prod_{\varphi,\psi:o}\prod_{d_1:(T\ (\supset\ \varphi\ \psi))}\prod_{d_2:(T\ \varphi)}(Cl\ \varphi\ d_2) \to (Cl\ (\supset\ \varphi\ \psi)\ d_1) \to (Cl\ \psi\ (\supset\text{-E}\ \varphi\ \psi\ d_1\ d_2)),$$
$$Cl_{\supset_\Box\text{-E}} : l\prod_{\varphi,\psi:o}\prod_{d_1:(T\ \Box(\supset\varphi\psi))}\prod_{d_2:(T\ \Box\varphi)}(Cl\ \Box(\supset\varphi\psi)\ d_1) \to (Cl\ \Box\varphi\ d_2) \to (Cl\ \Box\psi\ (\supset_\Box\text{-E}\ \varphi\ \psi\ d_1\ d_2))$$

$$\Box\text{-E} : \prod_{\varphi:o}(T\ \Box\varphi) \to (T\ \varphi),\quad Cl_{\Box\text{-E}} : \prod_{\varphi:o}\prod_{d:(T\ \Box\varphi)}(Cl\ \Box\varphi\ d) \to (Cl\ \varphi\ (\Box\text{-E}\ \varphi\ d));\quad \ldots$$

Figure 13.2: $\Sigma_{Cl}(\mathbf{N}K)$ and its extensions for $\mathbf{N}K4$, ...

Theorem 13.4 *The function* $\varepsilon_{X,\Delta}^{\Sigma_{Cl}(\mathbf{N}K)}$ *is a compositional bijection between proofs* $\pi : (\Delta \models_{\mathbf{N}K} \varphi$ *such that* $\mathrm{FV}(\pi) \subseteq X$*, and canonical terms* t*, such that* $\Gamma_X, \gamma_T(\Delta) \vdash_{\Sigma_{Cl}(\mathbf{N}K)} t{:}(T\ \varepsilon_X(\varphi))$.

As before, the proof of this theorem is composed by long and tedious inductions.

13.3 Encodings of special systems for $S4$

13.3.1 Boxed Judgement

In Figure 13.3 we give the signature $\Sigma_\Box(\mathbf{N}S4)$, which adopts a variant of the *Closed* technique for implementing Prawitz's first version of $\mathbf{N}S4$ [Pra65].

The idea is that the auxiliary judgement $Bx : \prod_{\varphi:o}(T\ \varphi) \to \mathsf{Prop}$ holds only on "boxed proof terms", that is, the terms which represent proofs depending only on boxed formulæ. At every moment, in the context For each assumption $a : (T\ \varphi)$ such that φ is boxed, there

Judgements

$$T : o \to \mathsf{Prop}$$
$$Bx : \prod_{\varphi:o} (T\varphi) \to \mathsf{Prop},$$

Axioms and Rules

$$\supset_\Box\text{-I} : \prod_{\varphi,\psi:o} \left(\prod_{d:(T\ \Box\varphi)} (Bx\ \Box\varphi\ d) \to (T\ \psi) \right) \to (T(\supset\ \ \Box\varphi\ \psi)),$$
$$\supset\text{-I} : \prod_{\varphi,\psi:o} ((T\ \varphi) \to (T\ \psi)) \to (T\ (\supset\ \ \varphi\ \psi)),$$
$$\supset\text{-E} : \prod_{\varphi,\psi:o} (T\ (\supset\ \ \varphi\ \psi)) \to (T\ \varphi) \to (T\ \psi),$$
$$\Box\text{-I} : \prod_{\varphi:o} \prod_{d:(T\ \varphi)} (Bx\ \varphi\ d) \to (T\ (\Box\varphi)),$$
$$\Box\text{-E} : \prod_{\varphi:o} (T\ (\Box\varphi)) \to (T\ \varphi),$$
$$Bx_{\supset_\Box\text{-I}} : \prod_{\varphi,\psi:o} \prod_{d:(\prod_{a:(T\ \Box\varphi)} (Bx\ \Box\varphi\ a) \to (T\ \psi))} \left(\prod_{a:(T\ \Box\varphi)} \prod_{b:(Bx\ \Box\varphi\ a)} (Bx\ \psi\ (d\ a\ b)) \right) \to (Bx\ (\supset\ \ \Box\varphi\ \psi)\ (\supset_\Box\text{-I}\ \varphi\ \psi\ d))$$
$$Bx_{\supset\text{-I}} : \prod_{\varphi,\psi:o} \prod_{d:(T\ \varphi) \to (T\ \psi)} \left(\prod_{a:(T\ \varphi)} (Bx\ \varphi\ a) \to (Bx\ \psi\ (da)) \right) \to (Bx\ (\supset\ \ \varphi\ \psi)\ (\supset\text{-I}\ \varphi\ \psi\ d)),$$
$$Bx_{\supset\text{-E}} : \prod_{\varphi,\psi:o} \prod_{d_1:(T\ (\supset\ \varphi\ \psi))} \prod_{d_2:(T\ \varphi)} (Bx\ (\supset\ \ \varphi\ \psi)\ d_1) \to (Bx\ \varphi\ d_2) \to (Bx\ \psi\ (\supset\text{-E}\ \varphi\ \psi\ d_1\ d_2))$$
$$Bx_{\Box\text{-I}} : \prod_{\varphi:o} \prod_{d:(T\ \varphi)} \prod_{b:(Bx\ \varphi\ d)} (Bx\ \Box\varphi\ (\Box\text{-I}\ \varphi\ d\ b)),$$
$$Bx_{\Box\text{-E}} : \prod_{\varphi:o} \prod_{d:(T\ \Box\varphi)} (Bx\ \Box\varphi\ d) \to (Bx\ \varphi\ (\Box\text{-E}\ \varphi\ d))$$

Figure 13.3: $\Sigma_\Box(\mathbf{N}S4)$.

is also an assumption $b_a : (Bx\ \varphi\ a)$ (see the encoding of contexts, below). Therefore, for establishing whether a proof is boxed, we need to look for such assumption for each free variable of the corresponding proof term. In the system there are rules for establishing the "boxed " judgement corresponding to each rule in $\mathbf{N}S4$. Additional rules for T can be induced by hypothesis-discharging rules, whenever the discharged formula is boxed (and hence belongs to the context). This is the case of $\supset_\Box$-E: we need to introduce in the context also the assumption $(Bx\ \Box\varphi\ d)$, which encodes the fact that the d assumes a boxed formula.

Given $\Delta \subseteq \Phi$ with $\mathrm{FV}(\Delta) \subseteq X$, we define the LF context $\gamma_\Box(\Delta)$ as follows:

$$\gamma_\Box(\Delta) \stackrel{\text{def}}{=} \begin{cases} \langle\rangle & \text{if } \Delta \equiv \emptyset \\ \gamma_\Box(\Delta'), v_\varphi : (T\varepsilon_X(\varphi)) & \text{if } \Delta \equiv \Delta', \varphi,\ \varphi \text{ is not boxed and } v_\varphi \text{ fresh for } \gamma_\Box(\Delta') \\ \gamma_\Box(\Delta'), v_\varphi : (T\varepsilon_X(\varphi)), & \text{if } \Delta \equiv \Delta', \varphi,\ \varphi \text{ is boxed and } v_\varphi, vb_\varphi \text{ fresh for } \gamma_\Box(\Delta') \\ \quad vb_\varphi : (Bx\varepsilon_X(\varphi)v_\varphi) & \end{cases}$$

The long proof of adequacy relies upon some very technical lemmata. We report here

only those needed for defining the encoding function. For sake of simplicity, we adopt the following definition: for p term and Γ context, we define

$$C(p,\Gamma) \stackrel{\text{def}}{=} \text{for all } v_\psi \in \mathrm{FV}(p),\ \text{if } (v_\psi{:}(T\ \varepsilon_X(\psi))) \in \Gamma \text{ then } (vb_\psi{:}(Bx\ \varepsilon_X(\psi)\ v_\psi)) \in \Gamma$$

Lemma 13.5 *Let p be a canonical term s.t. $\Gamma_X, \gamma_\Box(\Delta) \vdash_{\Sigma_\Box(\mathbf{N}S4)} p : (T\ t)$. If $C(p, \gamma_\Box(\Delta))$ holds then there is a canonical term b such that $\Gamma_X, \gamma_\Box(\Delta) \vdash_{\Sigma_\Box(\mathbf{N}S4)} b : (Bx\ t\ p)$.*

A consequence of this lemma is the existence of a function β_Δ which maps proof terms p whose free variables are "boxed," to proofs b of $(Bx\ \varphi\ p)$; this "reifies" the fact that p represents a proof which depends only on boxed assumptions. This function is inductively defined as follows.

$$\begin{array}{l}
\beta_\Delta(v_\varphi) \stackrel{\text{def}}{=} vb_\varphi \qquad\qquad \text{, if } \varphi \in \Delta \text{ and } \varphi \text{ boxed} \\
\beta_\Delta(\supset\text{-I}\ t\ t'(\lambda v_1{:}(T\ t).p)) \stackrel{\text{def}}{=} \\
\qquad (Bx_{\supset\text{-I}}\ t\ t'\ (\lambda v_1{:}(T\ t).p)(\lambda v_1{:}(T\ t)\lambda vb_1{:}(Bx\ t\ v_1).\beta_{\Delta,\delta_X(t)}(p))) \\
\beta_\Delta(\supset\text{-E}\ t\ t'\ p_1\ p_2) \stackrel{\text{def}}{=} (Bx_{\supset\text{-E}}\ t\ t'\ p_1\ p_2\ \beta_\Delta(p_1)\ \beta_\Delta(p_2)) \\
\beta_\Delta(\Box\text{-I}\ t\ p\ b) \stackrel{\text{def}}{=} (Bx_{\Box\text{-I}}\ t\ p\ b) \\
\beta_\Delta(\Box\text{-E}\ t\ p) \stackrel{\text{def}}{=} (Bx_{\Box\text{-E}}\ t\ p\ \beta_\Delta(p)) \\
\beta_\Delta(\supset_\Box\text{-I}\ t\ t'\ p_2) \stackrel{\text{def}}{=} (Bx_{\supset_\Box\text{-I}}\ t\ t'\ p_2(\lambda v_1{:}(T\ t)\lambda v_2{:}(Bx\ t\ v_1).\beta_{\Delta,\delta_X(A)}(p))) \\
\text{where } p_2 \stackrel{\text{def}}{=} \lambda v_1{:}(T\ t)\lambda v_2{:}(Bx\ t\ v_1).p
\end{array}$$

Lemma 13.6 *$\forall X, \Delta, \varphi$, if $\pi : (\Delta \vdash_{\mathbf{N}S4} \varphi)$ such that $\mathrm{FV}(\pi) \subseteq X$ then there exists a canonical form p such that $\Gamma_X, \gamma_\Box(\Delta) \vdash_{\Sigma_\Box(\mathbf{N}S4)} p : (T\ \varepsilon_X(\varphi))$.*

A consequence of this lemma is the existence of the function $\varepsilon^{\Sigma_\Box(\mathbf{N}S4)}_{X,\Delta}$, which maps proofs of $\mathbf{N}S4$ to canonical proof terms. This function is inductively defined as follows.

$$\begin{array}{l}
\varepsilon^{\Sigma_\Box(\mathbf{N}S4)}_{X,\Delta}(\varphi) \stackrel{\text{def}}{=} v_\varphi \qquad\qquad ,if\ \varphi \in \Delta \\
\varepsilon^{\Sigma_\Box(\mathbf{N}S4)}_{X,\Delta}(\supset\text{-I}_{\varphi\psi}(\pi')) \stackrel{\text{def}}{=} \begin{cases} (\supset_\Box\text{-I}\ \varepsilon_X(\varphi)\ \varepsilon_X(\psi) & \\ (\lambda v_\varphi{:}(T\varepsilon_X(\varphi))\lambda vb_\varphi{:}(Bx\varepsilon_X(\varphi)v_\varphi).\varepsilon^{\Sigma_\Box(\mathbf{N}S4)}_{X,\Delta,\varphi}(\pi'))) & \text{if } \varphi \text{ boxed} \\ (\supset\text{-I}\ \varepsilon_X(\varphi)\ \varepsilon_X(\psi)\ (\lambda v_\varphi{:}(T\varepsilon_X(\varphi)).\varepsilon^{\Sigma_\Box(\mathbf{N}S4)}_{X,\Delta,\varphi}(\pi'))) & \text{if } \varphi \neg \text{ boxed} \end{cases} \\
\varepsilon^{\Sigma_\Box(\mathbf{N}S4)}_{X,\Delta}(\supset\text{-E}_{\varphi,\psi}(\pi',\pi'')) \stackrel{\text{def}}{=} (\supset\text{-E}\ \varepsilon_X(\varphi)\ \varepsilon_X(\psi)\ \varepsilon^{\Sigma_\Box(\mathbf{N}S4)}_{X,\Delta}(\pi'')\ \varepsilon^{\Sigma_\Box(\mathbf{N}S4)}_{X,\Delta}(\pi')) \\
\varepsilon^{\Sigma_\Box(\mathbf{N}S4)}_{X,\Delta}(\Box\text{-I}_\varphi(\pi')) \stackrel{\text{def}}{=} (\Box\text{-I}\ \varepsilon_X(\varphi)\ \varepsilon_{X,\Delta}(\pi')\ \beta_\Delta(\varepsilon^{\Sigma_\Box(\mathbf{N}S4)}_{X,\Delta}(\pi'))) \\
\varepsilon^{\Sigma_\Box(\mathbf{N}S4)}_{X,\Delta}(\Box\text{-E}_\varphi(\pi')) \stackrel{\text{def}}{=} (\Box\text{-E}\ \varepsilon_X(\varphi)\ \varepsilon^{\Sigma_\Box(\mathbf{N}S4)}_{X,\Delta}(\pi'))
\end{array}$$

Theorem 13.7 *The function $\varepsilon^{\Sigma_\Box(\mathbf{N}S4)}_{X,\Delta}$ is a compositional bijection between proofs $\pi : (\Delta \vdash_{\mathbf{N}S4} \varphi$, such that $\mathrm{FV}(\pi) \subseteq X$ and canonical terms t, such that $\Xi(X), \gamma_\Box(\Delta) \vdash_{\Sigma_\Box(\mathbf{N}S4)} t : (T\ \varepsilon_X(\varphi))$.*

13.3.2 "Boxed Fringe"-judgement

For the sake of completeness we sketch here how to encode Prawitz's "third" system $\mathbf{N}S4_f$ [Pra65]. The signature $\Sigma_{Fr}(\mathbf{N}S4)$ appears in fig.13.4.

Judgements

$$T : o \to \mathsf{Prop},$$
$$BF : \prod_{\varphi:o}(T\ \varphi) \to \mathsf{Prop},$$

Axioms and Rules

$$\supset_{\Box}\text{-I} : \prod_{\varphi,\psi:o}\left(\prod_{d:(T\ \Box\varphi)}(BF\ \Box\varphi\ d) \to (T\ \psi)\right) \to (T\ (\supset\ \Box\varphi\ \psi)),$$
$$\supset\text{-I} : \prod_{\varphi,\psi:o}((T\ \varphi) \to (T\ \psi)) \to (T\ (\supset\ \varphi\ \psi)),$$
$$\supset\text{-E} : \prod_{\varphi,\psi:o}(T\ (\supset\ \varphi\ \psi)) \to (T\ \varphi) \to (T\ \psi),$$
$$\Box\text{-I} : \prod_{\varphi:o}\prod_{d:(T\ \varphi)}(BF\ \varphi\ d) \to (T\ \Box\varphi),$$
$$\Box\text{-E} : \prod_{\varphi:o}(T(\Box\varphi)) \to (T\ \varphi),$$
$$BF_{\supset_{\Box}\text{-I}} : \prod_{\varphi,\psi:o}\prod_{d:\prod_{a:(T\ \Box\varphi)}(BF\ \Box\varphi\ a)\to(T\ \psi)}\left(\prod_{a:(T\Box\varphi)}\prod_{b:(BF\ \Box\varphi\ a)}(BF\ \psi\ (d\ a\ b))\right) \to (BF\ (\supset\ \Box\varphi\ \psi)\ (\supset_{\Box}\text{-I}\ \varphi\ \psi\ d)),$$
$$BF_{\supset\text{-I}} : \prod_{\varphi,\psi:o}\prod_{d:(T\ \varphi)\to(T\ \psi)}\left(\prod_{a:(T\ \varphi)}(BF\ \varphi\ a) \to (BF\ \psi\ (d\ a))\right) \to (BF\ (\supset\ \varphi\ \psi)\ (\supset\text{-I}\ \varphi\ \psi\ d)),$$
$$BF_{\supset\text{-E}} : \prod_{\varphi,\psi:o}\prod_{d_1:(T\ (\supset\ \varphi\ \psi))}\prod_{d_2:(T\ \varphi)}(BF\ (\supset\ \varphi\ \psi)\ d_1) \to (BF\ \varphi\ d_2) \to (BF\ \psi\ (\supset\text{-E}\ \varphi\ \psi\ d_1\ d_2)),$$
$$BF'_{\supset\text{-E}} : \prod_{\varphi,\psi:o}\prod_{d_1:(T\ (\supset\ \varphi\ \Box\psi))}\prod_{d_2:(T\ \varphi)}(BF\ \Box\psi\ (\supset\text{-E}\ \varphi\ \Box\psi\ d_1\ d_2)),$$
$$BF_{\Box\text{-I}} : \prod_{\varphi:o}\prod_{d:(T\ \varphi)}\prod_{b:(BF\ \varphi\ d)}(BF\ \Box\varphi\ (\Box\text{-I}\varphi\ d\ b)),$$
$$BF_{\Box\text{-E}} : \prod_{\varphi:o}\prod_{d:(T\ \Box\varphi)}(BF\ \varphi\ (\Box\text{-E}\ \varphi\ d))$$

Figure 13.4: $\Sigma_{Fr}(\mathbf{N}S4)$.

The judgement $BF : \prod_{\varphi:o}(T\ \varphi) \to \mathsf{Prop}$ holds only on proofs with a fringe of boxed formulæ (in the minimal fragment of modal logic, boxed formulæ are all the essentially modal formulæ). In the system there are rules for establishing the "boxed fringe" judgement corresponding to each rule in $\mathbf{N}S4$. Additional rules for BF can be induced by elimination rules whenever the major premise (the eliminated formula) is boxed (and hence belongs to the fringe). This is the case, e.g., of $\supset$-E.

13.4 Encoding of multiple consequence relation systems

In this section we sketch briefly the encoding of multi-judgement systems $\mathbf{N}K'$, $\mathbf{N}K''$ (Sections 5.3.2, 5.3.3).

Judgements
$Ta, V : o \to \mathsf{Prop},$
Rules
$\supset\text{-I} : \prod_{\varphi,\psi:o} ((Ta\ \varphi) \to (Ta\ \psi)) \to (Ta(\supset\ \varphi\ \psi))$
$\Box_{Ta}\text{-I} : \prod_{\varphi:o} (Ta\ \varphi) \to (V\ \Box\varphi)$
$\supset\text{-E}_{Ta,Ta} : \prod_{\varphi,\psi:o} (Ta\ (\supset\ \varphi\ \psi)) \to (Ta\ \varphi) \to (Ta\ \psi)$
$\Box_{V}\text{-I} : \prod_{\varphi:o} (V\ \varphi) \to (V\ \Box\varphi),$
$\supset\text{-E}_{V,Ta} : \prod_{\varphi,\psi:o} (V\ (\supset\ \varphi\ \psi)) \to (Ta\ \varphi) \to (V\ \psi),$
$\supset\text{-E}_{Ta,V} : \prod_{\varphi,\psi:o} (Ta\ (\supset\ \varphi\ \psi)) \to (V\ \varphi) \to (V\ \psi),$
$\supset\text{-E}_{V,V} : \prod_{\varphi,\psi:o} (V\ (\supset\ \varphi\ \psi)) \to (V\ \varphi) \to (V\ \psi),$
$\supset_{\Box}\text{-E} : \prod_{\varphi,\psi:o} (Ta\ \Box(\supset\ \varphi\ \psi)) \to (Ta\ \Box\varphi) \to (Ta\ \Box\psi)$

$\Box\text{-E} : \prod_{\varphi:o} (Ta\ \Box\varphi) \to (Ta\ \varphi)$	$\Box_{\Box}\text{-I} : \prod_{\varphi:o} (Ta\ \Box\varphi) \to (Ta\ \Box\Box\varphi)$
$\Box_{\Diamond}\text{-I} : \prod_{\varphi:o} (Ta\ \Diamond\varphi) \to (Ta\ \Box\Diamond\varphi)$	$\Box_{\supset}\text{-E} : \prod_{\varphi:o} (Ta\ \Box(\supset\ (\Box\varphi)\ \varphi)) \to (Ta\ \Box\varphi)$

Figure 13.5: $\Sigma_{2j}(\mathbf{N}K')$ and its extensions for $\mathbf{N}K4'$,...

13.4.1 Encoding of $\mathbf{N}K'$ by two judgements

In Figure 13.5 we give the signature $\Sigma_{2j}(\mathbf{N}K')$ and its extension for systems $\mathbf{N}K4'$, $\mathbf{N}KT'$, ...

Given $\Delta \subseteq \Phi$ with $\mathrm{FV}(\Delta) \subseteq X$, we define the context $\gamma_{Ta}(\Delta)$ as follows:

$$\gamma_{Ta}(\Delta) \stackrel{\text{def}}{=} \begin{cases} \langle\rangle & \text{if } \Delta \equiv \emptyset \\ \gamma_{Ta}(\Delta'), v_\varphi : (Ta\ \varepsilon_X(\varphi)) & \text{if } \Delta \equiv \Delta', \varphi \text{ and } v_\varphi \text{ fresh for } \gamma_{Ta}(\Delta') \end{cases}$$

Theorem 13.8 *For $X \subset \Phi_a$, $\Delta \subseteq \Phi_X$, $\varphi \in \Phi_X$:*

- *There exists a compositional bijection between proofs $\pi : (\Delta \vdash \varphi)$, with $\mathrm{FV}(\pi) \subseteq X$, and canonical terms p, such that $\Gamma_X, \gamma_{Ta}(\Delta) \vdash_{\Sigma_{2j}(\mathbf{N}K)'} p : (Ta\ \varepsilon_X(\varphi))$.*

- *There exists a compositional bijection between proofs $\pi : (\Delta \Vdash \varphi)$, and canonical terms p, such that $\Gamma_X, \gamma_{Ta}(\Delta) \vdash_{\Sigma_{2j}(\mathbf{N}K)'} p : (V\ \varepsilon_X(\varphi))$.*

The proof appears in [AHMP97].

$$\Sigma_{3j}(\mathbf{N}K'') = \Sigma_{2j}(\mathbf{N}K') +$$

Judgements
$T : o \to \mathsf{Prop}$,
Axioms and Rules
$C : \prod_{\varphi:o}(V\ \varphi) \to (T\ \varphi)$
$\supset_T$-I : $\prod_{\varphi,\psi:o}((T\ \varphi) \to (T\ \psi)) \to (T\ (\supset\ \varphi\ \psi))$
$\supset_T$-E : $\prod_{\varphi,\psi:o}(T\ (\supset\ \varphi\ \psi)) \to (T\ \varphi) \to (T\ \psi)$,
... similarly for negation and *ff*.

$\Box$-E : $\prod_{\varphi:o}(Ta\ \Box\varphi) \to (Ta\ \varphi)$	$\Box_\Box$-I : $\prod_{\varphi:o}(Ta\ \Box\varphi) \to (Ta\ \Box\Box\varphi)$
$\Box_\Diamond$-I : $\prod_{\varphi:o}(Ta\ \Diamond\varphi) \to (Ta\ \Box\Diamond\varphi)$	$\Box_\supset$-E : $\prod_{\varphi:o}(Ta\ \Box(\supset\ (\Box\varphi)\ \varphi)) \to (Ta\ \Box\varphi)$

Figure 13.6: $\Sigma_{3j}(\mathbf{N}K'')$ and its extensions for $\mathbf{N}K4''$,....

13.4.2 Encoding of $\mathbf{N}K''$ by three judgements

In order to encode the system $\mathbf{N}K''$, we add to $\Sigma_{2j}(\mathbf{N}K'')$ a judgement $T : o \to \mathsf{Prop}$, whose constructors are like those of Ta plus a constant C which represents the EMBED' rule (Figure 13.6). We can prove then the adequacy for the "third" level of the system:

Theorem 13.9 *There is a compositional bijection between proofs* $\pi : (\Delta \Vdash_{\mathbf{N}K''} \varphi)$ *with* $\mathrm{FV}(\pi) \subseteq X$ *and canonical terms* t *such that* $\Gamma_X, \gamma_T(\Delta) \vdash_{\Sigma_{3j}(\mathbf{N}K'')} t : (T\ \varepsilon_X(\varphi))$.

Chapter 14

Encoding of Dynamic Logics

In this chapter we present the encoding of the systems for Propositional Dynamic Logic and First-Order Dynamic Logic, presented in Chapters 6, 7.

14.1 Encoding of *PDL*

14.1.1 Encoding of the language

The encoding of the language of *PDL* follows simply the paradigmatical methodology (Chapter 11). Each syntactic category is represented by a type, and each syntactic constructor is represented by a functional constant (Figure 14.1). There is also a function $b2p : B \to P$, defined by induction on the syntax of box-free propositions, which embeds box-free propositions into propositions. When clear from the context, it will be omitted for sake of readability. Actually, applications of such a function are computable (`Simpl`ifable) in the Coq environment.

Let $\xi : B \cup Prog \cup \Phi \to B \cup C \cup P$ be the obvious compositional bijective representation of propositions. For the sake of simplicity, ξ will be often omitted; therefore, with the same term we will denote a formula as well as its encoding in the LF signature; similarly we shall deal with sets of assumptions.

14.1.2 Encoding of the finitary $\mathbf{N}_f PDL$

Since $\mathbf{N}_f PDL$ is in ND-style, most of the rules are encoded straightforwardly following the methodology of [HHP93, AHMP92] (Figure 14.2). The main difficulty is the representation of the modal rule $^*_f$-I. Here we adopt the *world technique* presented in Section 13.2.1.

Theorem 14.1 (Adequacy and Faithfulness for $\mathbf{N}_f PDL$) *Let Γ range over finite sets of assumptions, d over canonical proof terms of type $(T\ w\ \varphi)$ for some φ, w. Then, $\forall \Gamma, \varphi$:*

$$\Gamma \vdash_{\mathbf{N}_f PDL} \varphi \iff \exists d.\ \gamma_w(\Gamma) \vdash_{\Sigma(PDL)} d : (T\ w\ \varphi).$$

14.1.3 Encoding of $\mathbf{N}PDL$

The encoding of the infinitary proof system $\mathbf{N}PDL$ is very close to the one presented in 14.2. The only difference is the representation of the infinitary rule *-I. The system $\mathbf{N}PDL$

$$\begin{array}{lcl@{\qquad}lcl@{\qquad}lcl}
B & : & Set & C & : & Set & P & : & Set \\
\neg_b & : & B \to B & {}^* & : & C \to C & \neg & : & P \to P \\
\supset_b, \wedge_b & : & B \to B \to B & ? & : & B \to C & \supset, \wedge & : & P \to P \to P \\
 & & & ;, + & : & C \to C \to C & [\cdot]\cdot & : & C \to P \to P
\end{array}$$

Figure 14.1: Representation of $\mathcal{L}_{PDL}$ in $\Sigma(PDL)$.

can take into account infinite sets of assumptions, hence we need to be able to refer to infinite sets of formulæ. This we achieve by means of a recursive function from naturals to (proofs of) propositions, as follows:

$$^*\text{-I} \;:\; \prod_{p:P}\prod_{c:C}\prod_{w:U}\left(\prod_{n:nat}(T\ w\ (I\ c\ p\ n))\right) \to (T\ w\ [c^*]\,p)$$

The version of the rule *-I, that we encode in Coq is therefore as follows:

$$^*\text{-I}\ \frac{\text{for all } n \in \mathcal{N} : I(c,\varphi,n)}{[c^*]\,\varphi} \qquad \text{where} \qquad \begin{array}{l} I : Prog \to \Phi \to \mathcal{N} \to \Phi \\ I(c,\varphi,0) \stackrel{\text{def}}{=} \varphi, \quad I(c,\varphi,n+1) \stackrel{\text{def}}{=} [c]\,I(c,\varphi,n) \end{array}$$

In Coq the premise of the rule is represented using a λ-term of type $nat \to Prop$. Thus, in the encoding, we can refer only to lists of premises which can be enumerated by a function provably total in PA^w. Nevertheless this is enough for dealing with representable lists of assumptions.

Theorem 14.2 (Adequacy and Faithfulness for *PDL*) *Let Γ range over representable sets of assumptions, d over canonical proof terms of type $(T\ w\ \varphi)$ for some φ, w. Then $\forall \Gamma, p. \Gamma \vdash_{\mathbf{N}PDL} p \iff \exists d.\ \gamma_w(\Gamma) \vdash_{\Sigma(PDL)} d : (T\ w\ p)$.*

14.2 Encoding of First Order Dynamic Logic

This system will extend those presented in the previous Section 14.1.

14.2.1 Encoding the language of *DL*

As we have discussed in Section 11.1, the presence of identifiers in formulæ standing for left-hand values which cannot be substituted for, forces us to introduce a specific type for identifiers. Therefore, substitutions of terms for identifiers cannot be handled any more "for free" by the metalanguage, using *higher order syntax*. Nevertheless, we can still handle at the metalevel substitution of identifiers for identifiers, as we have described in Sections 11.1, 11.3. The encoding of the language of *DL* (Figure 14.3) extends that of *PDL* (Figure 14.1).

14.2.2 The assignment rules

As remarked earlier, we cannot exploit higher-order syntax directly to encode $()[t/x]$, the substitution operator, as was possible in [AHMP92, HHP93, MP91]. The naïve encoding of the assignment constructor, $:=: Te \to Te \to C$, could yield meaningless commands

$$\wedge\text{-I} \;:\; \prod_{p,q:P}\prod_{w:U}(T\ w\ p)\to(T\ w\ q)\to(T\ w\ (p\wedge q))$$

$$\wedge\text{-E}_\text{L} \;:\; \prod_{p,q:P}\prod_{w:U}(T\ w\ (p\wedge q))\to(T\ w\ p)$$

$$\wedge\text{-E}_\text{R} \;:\; \prod_{p,q:P}\prod_{w:U}(T\ w\ (p\wedge q))\to(T\ w\ q)$$

$$\supset\text{-I} \;:\; \prod_{p,q:P}\prod_{w:U}((T\ w\ p)\to(T\ w\ q))\to(T\ w\ (p\supset q))$$

$$\supset\text{-E} \;:\; \prod_{p,q:P}\prod_{w:U}(T\ w\ (p\supset q))\to(T\ w\ p)\to(T\ w\ q)$$

$$;\text{-I} \;:\; \prod_{p:P}\prod_{c_1,c_2:C}\prod_{w:U}(T\ w\ \ [c_1]\,[c_2]\,p)\to(T\ w\ \ [c_1;c_2]\,p)$$

$$;\text{-E} \;:\; \prod_{p:P}\prod_{c_1,c_2:C}\prod_{w:U}(T\ w\ \ [c_1;c_2]\,p)\to(T\ w\ \ [c_1]\,[c_2]\,p)$$

$$+\text{-I} \;:\; \prod_{p:P}\prod_{c_1,c_2:C}\prod_{w:U}(T\ w\ \ [c_1]\,p)\to(T\ w\ \ [c_2]\,p)\to(T\ w\ \ [c_1+c_2]\,p)$$

$$+\text{-E}_\text{L} \;:\; \prod_{p:P}\prod_{c_1,c_2:C}\prod_{w:U}(T\ w\ \ [c_1+c_2]\,p)\to(T\ w\ \ [c_1]\,p)$$

$$+\text{-E}_\text{R} \;:\; \prod_{p:P}\prod_{c_1,c_2:C}\prod_{w:U}(T\ w\ \ [c_1+c_2]\,p)\to(T\ w\ \ [c_2]\,p)$$

$$?\text{-I} \;:\; \prod_{p:P}\prod_{b:B}\prod_{w:U}((T\ w\ b)\to(T\ w\ p))\to(T\ w\ \ [b?]\,p)$$

$$?\text{-E} \;:\; \prod_{p:P}\prod_{b:B}\prod_{w:U}(T\ w\ \ [b?]\,p)\to(T\ w\ b)\to(T\ w\ p)$$

$$\overset{*}{f}\text{-I} \;:\; \prod_{p:P}\prod_{c:C}\prod_{w:U}\left(\prod_{w':U}(T\ w'\ p)\to(T\ w'\ [c]p)\right)\to(T\ w\ p)\to(T\ w\ \ [c^*]\,p)$$

$$^*\text{-E} \;:\; \prod_{p:P}\prod_{c:C}\prod_{w:U}(T\ w\ \ [c^*]\,p)\to\prod_{n:nat}(T\ w\ (I\ c\ p\ n))$$

where $(I\ c\ p\ 0)=p,\quad (I\ c\ p\ (S\ n))=[c]\,(I\ c\ p\ n)$

Figure 14.2: Representation of $\mathbf{N}_f PDL$ in the signature $\Sigma(PDL)$.

X, Te, B, C, P	:	Set	$isId$	:	$X \to Te$	$[\cdot]\cdot$	:	$C \to P \to P$
$\neg_b$	:	$B \to B$	$0, 1$	:	Te	*	:	$C \to C$
$\supset_b, \wedge_b$	:	$B \to B \to B$	$+, *$	:	$Te \to Te \to Te$	$?$	:	$B \to C$
$\neg$	:	$P \to P$	$=_b, <_b$	:	$Te \to Te \to B$	$;, +$	:	$C \to C \to C$
$\supset, \wedge$	:	$P \to P \to P$	$=, <$	:	$Te \to Te \to P$	$:=$	:	$X \to Te \to C$

Figure 14.3: Representation of $\mathcal{L}(DL)$ in $\Sigma(DL)$ (some constructors).

$$:=\text{-I}: \prod_{A:X\to P} \prod_{x:X} \prod_{t:Te} \prod_{w:U} \left(\prod_{y:X} (isnotin\ y\ P\ \forall A) \to (isnotin\ y\ Te\ t) \to (T\ w\ (y = t)) \to (T\ w\ (A\ y)) \right) \\ \to (isnotin\ x\ P\ \forall A) \to (T\ w\ ([x := t](A\ x)))$$

$$:=\text{-E}: \prod_{A:X\to P} \prod_{q:P} \prod_{x:X} \prod_{t:Te} \prod_{w:U} \left(\prod_{y:X} (isnotin\ y\ P\ \forall A) \to (isnotin\ y\ Te\ t) \to (isnotin\ y\ P\ q) \to \right. \\ \left. (T\ w\ (y = t)) \to (T\ w\ (Ay)) \to (T\ w\ q)) \to (isnotin\ x\ P\ \forall A) \to \right. \\ \left. (T\ w\ ([x := t]\,(A\ x))) \to (T\ w\ q) \right.$$

Figure 14.4: The LF encoding of the rules for assignment.

such as $0 := 1$. Substitution has to be dealt with differently from [HHP93], rather in the style of [BH90]. The encodings of the rules :=-I and :=-E appear in Figure 14.4. We need to express the fact that an identifier is "fresh", i.e. that it is different from any other pre-existing identifier. To this end, we generalize Mason's idea [AHMP92] later expounded in [BH90, Mic94], and we introduce the two auxiliary judgements, $isin, isnotin : X \to \prod_{A:Set} A \to Prop$. The intuitive meaning of $(isin\ x\ A\ a)$ is "the identifier x appears in the phrase a whose type is A;" dually for *isnotin*. These two judgements are derivable by means of a simple set of rules which are polymorphic in the syntactic constructors (Figure 14.5). The inference of these judgements is completely syntax-driven: it is sufficient to look at the top-level constructor of the phrase for deciding which rule has to be applied. The premise $(isnotin\ x\ P\ \forall A)$ of the :=-I rule enforces the fact that the context $A(\cdot)$ does not contain any occurrence of x. In both rules we have also to reify the "freshness condition" of variables locally quantified in premises. This is achieved by assuming suitable *isnotin* judgements. Such reified assumptions are needed to deal with "contexts" such as $A(\cdot)$ above, or the CongrId rule below.

14.2.3 The congruence rules

The encodings of Congr and CongrId appear in Figure 14.6. In encoding CongrId, we have to check that the context $A(\cdot)$ does not contain any occurrence of x, y. This is enforced as for :=-I, :=-E, via the premises $(isnotin\ x\ P\ \forall A)$ and $(isnotin\ y\ P\ \forall A)$ In encoding Congr we have to check that the predicate A is command-free. This is easily

$$
\begin{aligned}
isin\_x &: \prod_{x:X}(isin\ x\ X\ x)\\
isin\_1 &: \prod_{x:X}\prod_{s_1,s_2:Set}\prod_{op:s_1\to s_2}\prod_{p:s_1}(isin\ x\ s_1\ p)\to(isin\ x\ s_2\ (op\ p))\\
isin\_2l &: \prod_{x:X}\prod_{s_1,s_2,s_3:Set}\prod_{op:s_1\to s_2\to s_3}\prod_{p_1:s_1}\prod_{p_2:s_2}(isin\ x\ s_1\ p_1)\to(isin\ x\ s_3\ (op\ p_1\ p_2))\\
isin\_2r &: \prod_{x:X}\prod_{s_1,s_2,s_3:Set}\prod_{op:s_1\to s_2\to s_3}\prod_{p_1:s_1}\prod_{p_2:s_2}(isin\ x\ s_2\ p_2)\to(isin\ x\ s_3\ (op\ p_1\ p_2))\\
isin\_n &: \prod_{s_1,s_2:Set}\prod_{op:(X\to s_1)\to s_2}\prod_{p:X\to s_1}\left(\prod_{y:X}(isin\ x\ s_1\ (p\ y))\right)\to(isin\ x\ s_2\ (op\ p))
\end{aligned}
$$

$$
\begin{aligned}
isnotin\_symm &: \prod_{x,y:X}(isnotin\ y\ X\ x)\to(isnotin\ x\ X\ y)\\
isnotin\_zero &: \prod_{x:X}(isnotin\ x\ Te\ zero) \qquad isnotin\_false : \prod_{x:X}(isnotin\ x\ P\ false)\\
isnotin\_1 &: \prod_{x:X}\prod_{s_1,s_2:Set}\prod_{op:s_1\to s_2}\prod_{p:s_1}(isnotin\ x\ s_1\ p)\to(isnotin\ x\ s_2\ (op\ p))\\
isnotin\_2 &: \prod_{x:X}\prod_{s_1,s_2,s_3:Set}\prod_{op:s_1\to s_2\to s_3}\prod_{p:s_1}\prod_{p:s_2}\\
&\quad (isnotin\ x\ s_1\ p_1)\to(isnotin\ x\ s_2\ p_2)\to(isnotin\ x\ s_3\ (op\ p_1\ p_2))\\
isnotin\_el &: \prod_{x,y:X}\prod_{s:Set}\prod_{p:s}(isnotin\ x\ s\ p)\to(isin\ y\ s\ p)\to(isnotin\ y\ X\ x)\\
isnotin\_n &: \prod_{s_1,s_2:Set}\prod_{op:(X\to s_1)\to s_2}\prod_{p:X\to s_1}\left(\prod_{y:X}(isnotin\ x\ X\ y)\to(isnotin\ x\ s_1\ (p\ y))\right)\\
&\quad \to(isnotin\ x\ s_2\ (op\ p))
\end{aligned}
$$

Figure 14.5: The rules for auxiliary judgements *isin*, *isnotin* of $\Sigma(DL)$.

$$\textsc{CongrId} : \prod_{x,y:X} \prod_{A:X\to P} \prod_{w:U} (isnotin\ x\ P\ \forall A) \to (isnotin\ y\ P\ \forall A) \to (T\ w\ (A\ x)) \to (T\ w\ ((isId\ x) = (isId\ y))) \to (T\ w\ (A\ y))$$

$$\textsc{Congr} : \prod_{t_1,t_2:Te} \prod_{A:Te\to P} \prod_{w:U} (T\ w\ (A\ t_1)) \to (T\ w\ (t_1 = t_2)) \to (BF\ (A\ t_2)) \to (T\ w\ (A\ t_2))$$

Figure 14.6: The LF encoding of the congruence rules.

$$\forall\text{-I} : \prod_{A:X\to P} \prod_{w:U} \left(\prod_{x:X} (isnotin\ x\ P\ \forall A) \to (T\ w\ (A\ x)) \right) \to (T\ w\ \forall A)$$

$$\forall\text{-E} : \prod_{A:X\to P} \prod_{q:P} \prod_{t:Te} \prod_{w:U} \left(\prod_{x:X} (isnotin\ x\ Te\ t) \to (isnotin\ x\ P\ q) \to (isnotin\ x\ P\ \forall A) \to (T\ w\ (x = t)) \to (T\ w\ (A\ x)) \to (T\ w\ q)) \to (T\ w\ \forall A) \to (T\ w\ q) \right.$$

Figure 14.7: The LF encoding of the ∀-I, ∀-E rules.

achieved by introducing a new judgement $BF : P \to Prop$, whose rules are the following:

$BF\_false : (BF\ false)$	$BF\_forall : \prod_{p:X\to P} (\prod_{x:X} (BF\ (p\ x))) \to (BF\ (\forall p)$
$BF\_eq : \prod_{t_1,t_2:Te} (BF\ (t_1 = t_2))$	$BF\_and : \prod_{p,q:P} (BF\ p) \to (BF\ q) \to (BF\ (p \wedge q))$
$BF\_not : \prod_{p:P} (BF\ p) \to (BF\ (\neg p))$	$BF\_imp : \prod_{p,q:P} (BF\ p) \to (BF\ q) \to (BF\ (p \supset q))$

Clearly, derivations of *BF* are syntax-driven and can be mostly automated in the Coq environment using the *Auto* tactic.

14.2.4 The ∀-quantifier rules

The encoding of the rules for ∀ appearing in Figure 14.7, is not as straightforward as in the standard FOL case. We have to deal with side-conditions and reify "freshness" assumptions on the variables locally quantified in premises, as was the case for the :=-I and :=-E rules.

14.3 Adequacy of the encoding

The statement of the Adequacy Theorem for the encoding $\Sigma(DL)$ is more problematic than in the "paradigm case" of FOL [HHP93], since we have to take into account infinite sets of formulæ. Clearly, this cannot be done in full generality and we will be able to state the Adequacy Theorem only with respect to *representable* sets of assumptions, i.e. sets of formulæ whose encodings can be enumerated in Coq. Formally, $\Gamma = \{p_n \mid n \in \mathcal{N}\}$ is *representable (in a context* Δ*)* if there exists a term G such that $\Delta \vdash_{\Sigma(DL)} G : nat \to P$ and for all $n \in \mathcal{N} : \Delta \vdash_{\Sigma(DL)} (G\ \overline{n}) = \xi(p_n)$

Given a representable set of assumptions Γ, in order to define $\gamma(\Gamma)$, the *Coq representation of* Γ, we proceed as follows. First of all, we assume, for each free identifier appearing in Γ, the identifier itself and the judgement asserting that it is different from any other identifier (notice that, for obvious reasons, we are interested in considering only a finite set of identifiers at any given time); we put

$$\iota(\Gamma) \stackrel{\text{def}}{=} \{x : X \mid x \in \mathrm{FV}(\Gamma)\} \cup \{i_{xy} : (\mathit{isnotin}\ x\ X\ y) \mid x, y \in \mathrm{FV}(\Gamma), x \neq y\}$$

If $\Gamma = \{p_1, \ldots, p_n\}$ is finite then we put

$$\gamma(\{p_1, \ldots, p_n\}) = \iota(\Gamma) \cup \{w : U, u_1 : (T\ w\ \xi(p_1)), \ldots, u_n : (T\ w\ \xi(p_n))\}$$

Otherwise, if $\Gamma = \{p_n \mid n \in \mathcal{N}\}$ is infinite and representable by a term G in $\iota(\Gamma)$, we put

$$\gamma(\Gamma) = \iota(\Gamma) \cup \{w : U, a : \prod_{n:nat} (T\ w\ (G\ n))\}.$$

Thus we have the following theorem, which is proved by induction.

Theorem 14.3 (Adequacy of $\Sigma(DL)$) *Let Γ be a representable (in $\iota(\Gamma)$) set of assumptions. Then*

1. $\forall \Gamma$, *if* $\gamma(\Gamma) \vdash M : A$, *where* $A \in \{X, Te, B, C, P\}$, *then*

$$\begin{array}{rcl} (\exists u.\gamma(\Gamma) \vdash_{\mathbf{N}DL} u : (\mathit{isin}\ x\ A\ M)) & \iff & x \in \mathrm{FV}(M) \\ (\exists u.\gamma(\Gamma) \vdash_{\mathbf{N}DL} u : (\mathit{isnotin}\ x\ A\ M)) & \iff & x \notin \mathrm{FV}(M) \end{array}$$

2. $\forall \Gamma, p : \quad \Gamma \vdash_{\mathbf{N}DL} p \iff \forall w. \exists d.\ \gamma(\Gamma) \vdash_{\Sigma(DL)} d : (T\ w\ p)$.

Proof. (Hint) The two properties have to be proved by simultaneous induction:

$\Rightarrow$ on the height of derivations in $\mathbf{N}DL$

$\Leftarrow$ on the syntax of canonical terms $\square$

Chapter 15

Encoding of μ-calculus

In this chapter we present the encoding of the μ-calculus, presented in Chapter 9. The proof system does present new difficulties, with respect to those of Modal and Dynamic Logics, since it is finitary and has just one proof rule. The main issues are instead related to the representation of the syntax of μ-formulæ.

15.1 Encoding of the language

The encoding of the language of μ-calculus is quite elaborate. The first problem is the presence of a negative formula constructor (the μ); as we have already pointed out in Section 11.2, this kind of constructors cannot be used in inductive definitions of CIC. In the case of μ-calculus, the substitution we have to implement is the same of the metalanguage, hence we can apply the bookkeeping technique implemented by means of Leibnitz equality (Section 11.3).

The second problem is the presence of a context-sensitive condition on the applicability of μ. So far, as described in Section 11.4.1, the only way for enforcing the context-sensitive condition is to use a Σ-type, that is, to define a subtype.

We introduce a separate type, *var*, for the identifiers. There are no constructors for this types: we only assume that there are infinite many variables.

```
Parameter var : Set.
Axiom var_nat : (Ex [srj:var->nat](n:nat)(Ex [x:var](srj x)=n)).
```

Then, we define the set of preformulæ of μ-calculus, also those not well formed:

```
Parameter Act : Set.
Inductive o   : Set :=
          ff  : o
        | Not : o -> o
        | And : o -> o -> o
        | Imp : o -> o -> o
        | Box : Act -> o -> o
        | Var : var -> o
        | mu  : (var->o) -> o.
```

Since we have not declared *var* as an inductive set, there are no exotic terms.

Now, in order to define the subtype of well formed formulæ, we need to formalize the system for positivity/negativity presented in Figure 9.2. Therefore, we can define two judgements on preformulæ,

$$posin, negin : var \to o \to \mathsf{Prop}$$

A careful analysis of the proof system (Figure 9.2) points out that the derivation of these judgement is completely syntax driven. It is therefore natural to define these judgements as recursively defined functions, instead of inductively defined propositions. This is indeed possible, but the rules for the binding operators introduce an implicit quantification over the set of variables *different from the one we are looking for*. This quantification is rendered by assuming a locally new variable (y) and that it is different from the variable x (see last cases):

```
Fixpoint posin [x:var;A:o] : Prop :=
   <Prop>Case A of
               True
               [B:o](negin x B)
               [A1,A2:o](posin x A1)/\(posin x A2)
               [A1,A2:o](negin x A1)/\(posin x A2)
               [a:Act][A1:o](posin x A1)
               [y:var]True
               [F:var->o](y:var)~(x=y) -> (posin x (F y))
          end
with negin [x:var;A:o] : Prop :=
   <Prop>Case A of
               True
               [B:o](posin x B)
               [A1,A2:o](negin x A1)/\(negin x A2)
               [A1,A2:o](posin x A1)/\(negin x A2)
               [a:Act][A1:o](negin x A1)
               [y:var]~(x=y)
               [F:var->o](y:var)~(x=y) -> (negin x (F y))
          end.
```

Therefore, in general a goal (*posin x A*) can be simplified to a conjunction of only three forms of propositions: $True$, negations of equalities or implications from negations of equalities to another conjunction of the same form. These three forms are dealt simply in the Coq environment, hence proving this kind of goals is a simple task.

Finally, we can define when a preformula is well formed: when every application of μ satisfies the positivity condition:

```
Fixpoint iswf [A:o] : Prop :=
   <Prop>Case A of
               True
               [A1:o](iswf A1)
               [A1:o][A2:o](iswf A1)/\(iswf A2)
               [A1:o][A2:o](iswf A1)/\(iswf A2)
```

```
              [a:Act][A1:o](iswf A1)
              [x:var]True
              [F:var->o](x:var)
                      ((notin x (mu F)) -> (posin x (F x)))/\(iswf (F x))
        end.
```

Hence, each formula of the μ-calculus is represented by a pair preformula-proof of its well-formness:

```
(* the set of well formed formuale *)
Record wfo : Set := mkwfo {
             prp : o;
             cnd : (iswf prp)
             }.
```

Theorem 15.1 *There is a bijection between well-formed formulæ of the μ-calculus and canonical forms of type wfo*

Proof. (Sketch) Long but not difficult inductions. First, we prove that *posin*, *negin* adequately represent the positivity/negativity proof system. Due to its structure, it is easy to prove that the proposition *posin x A* is inhabited by at most one canonical form (that is to say, there is at most one way for proving that a preformula is well-formed). Therefore, a preformula φ is a formula iff each application of μ is valid, iff for each application of μ there exists a (unique) witness of *posin*, iff there exists a (unique) inhabitant of *iswf* φ. □

15.2 Encoding of proof system

Finally we examine the implementation of the bookkeeping by means of Leibnitz equality: in the recursion rules,

```
Axiom mu_I   : (F:var->o)
               ((z:var)(notin z (mu F)) -> (Var z)=(mu F) -> (T w (F z)))
               -> (T w (mu F)).

Axiom mu_E   : (F:var->o)(iswf A) ->
               ((z:var)(notin z (mu F)) -> (Var z)=A ->
                       (w':U)(T w' (F z)) -> (T w' A))
               ->
               (T w (mu F)) -> (T w A).
```

we locally assume

1. a new variable, z;
2. the fact that z does not appear in the formula, i.e. it is *fresh*;
3. the binding between z and the formula itself, for successive unfoldings

The judgement *notin* (and the dual *isin*) are auxiliary judgements for occur-checking; they may be needed in the rest of derivation for inferring well-formness of discharged formulæ in rules RAA, $\supset$-I, $\neg$-I.

Part IV

Pragmatics

Chapter 16

The Implementation of $\mathbf{N}_f DL$ and $\mathbf{N}DL$

16.1 The signature ΣDL

This implementation subsumes the one of ΣPDL, of course.

```
(*      Full Dynamic Logic                                      *)
(*      Modal technique: World Parameter                        *)
(*      Subst technique: isin, isnotin                          *)
(*      Quant technique: Higher Order Syntax                    *)

(*--------------------------------------------------------------*)
(*                          SYNTACTIC CATEGORIES                *)
(*--------------------------------------------------------------*)

(* Set of Identifiers: it may be any enumerable set of objects *)
Parameter X : Set.
Parameter i : nat -> Set.
(* hence, (i n) represents the n-th identifier *)

(* Set of Terms *)
Inductive Te : Set
        := isX   : X -> Te
|  zero  : Te
|  succ  : Te -> Te
|  pls   : Te -> Te -> Te.

(* Set of Box-Quantifier-Free Formulae *)
Inductive B : Set
:= bff : B
|  bEq    : Te -> Te -> B
|  bNot   : B -> B
|  bAnd   : B -> B -> B
```

```
|  bImp   : B -> B -> B.

(* Set of Regular Programs *)
Inductive C : Set
:= ass   : X -> Te -> C
|  comp  : C -> C -> C
|  test  : B -> C
|  iter  : C -> C
|  ndc   : C -> C -> C. (* Non Deterministic Choice *)

Definition ifte : B -> C -> C -> C
:= [b:B][c1:C][c2:C]
(ndc (comp (test b) c1) (comp (test (bNot b)) c2)).

Definition while : B -> C -> C
 := [b:B][c:C]
 (comp (iter (comp (test b) c)) (test (bNot b))).

(* Set of All Formulae *)
Inductive P : Set
:= false  : P
| Eq     : Te -> Te -> P
| Not    : P -> P
| And    : P -> P -> P
| Imp    : P -> P -> P
| box    : C -> P -> P
| forall : (X -> P) -> P.

(* embedding function from box-quantifier free to all formulae *)
Fixpoint b2p [b:B] : P
 := <P>Case b of
 (* bff    *) false
         (* bEq    *) [t1,t2:Te](Eq t1 t2)
         (* bNot   *) [b1:B](Not (b2p b1))
         (* bAnd   *) [b1,b2:B](And (b2p b1) (b2p b2))
         (* bImp   *) [b1,b2:B](Imp (b2p b1) (b2p b2))
end.


(*-------------------------------------------------------------*)
(*                        JUDGMENTS                            *)
(*-------------------------------------------------------------*)

(* The Universe: auxiliary set of Worlds *)
Parameter U : Set.
```

```
(************************)
(* The proving judgment *)
(************************)
Parameter T : U -> P -> Prop.

(*************************************************************)
(* auxiliar occur-check judgments                            *)
(*************************************************************)
Inductive isin [x:X] : (A:Set) A -> Prop
:= isin_x : (isin x X x)
| (* 1-arity constructors *)
  isin_1 : (s1,s2:Set)(op: s1->s2)(p:s1)
   (isin x s1 p) ->
   (isin x s2 (op p))
| (* 2-arity constructors *)
  isin_2l : (s1,s2,s3:Set)(op: s1->s2->s3)(p1:s1)(p2:s2)
    (isin x s1 p1) ->
    (isin x s3 (op p1 p2))
| isin_2r : (s1,s2,s3:Set)(op: s1->s2->s3)(p1:s1)(p2:s2)
    (isin x s2 p2) ->
    (isin x s3 (op p1 p2))
| isin_n  : (s1,s2:Set)(op: (X->s1)->s2)(p:X->s1)
    ((y:X)(isin x s1 (p y))) ->
    (isin x s2 (op p)).
Hint isin_x.

(* due to the negative case, isnotin cannot be defined inductively *)
Parameter isnotin : X -> (A:Set) A -> Prop.
Axiom isnotin_base : (x,y:X) ~(x=y) -> (isnotin x X y).

(* 0-arity constructors *)
Axiom isnotin_zero : (x:X)(isnotin x Te zero).
Axiom isnotin_false: (x:X)(isnotin x P false).
(* 1-arity constructors *)
Axiom isnotin_1    : (x:X)(s1,s2:Set)(op:s1->s2)(p:s1)
     (isnotin x s1 p) -> (isnotin x s2 (op p)).
(* 2-arity constructors *)
Axiom isnotin_2    : (x:X)(s1,s2,s3:Set)(op:s1->s2->s3)(p1:s1)(p2:s2)
     (isnotin x s1 p1) ->
     (isnotin x s2 p2) ->
     (isnotin x s3 (op p1 p2)).
(* negative constructors on X (forall) *)
Axiom isnotin_n    : (x:X)(s1,s2:Set)(op:(X->s1)->s2)(p:X->s1)
     ((y:X)(isnotin x X y) -> (isnotin x s1 (p y))) ->
     (isnotin x s2 (op p)).
(* elimination (separation) *)
```

```
Axiom isnotin_el   : (s:Set)(p:s)(x,y:X)
     (isnotin x s p) ->
     (isin y s p) ->
     (isnotin y X x).
Hint isnotin_zero isnotin_false.

Lemma isnotin_symm : (x,y:X)(isnotin y X x) -> (isnotin x X y).
Intros; Apply isnotin_el with p:=x; Auto.
Qed.

(*************************************************************)
(* auxiliar judgments for the box-free condition           *)
(*************************************************************)
Inductive BF : P -> Prop
:= BF_false  : (BF false)
| BF_eq     : (t1,t2:Te)(BF (Eq t1 t2))
| BF_not    : (p:P)(BF p) -> (BF (Not p))
| BF_and    : (p,q:P)(BF p) -> (BF q) -> (BF (And p q))
| BF_imp    : (p,q:P)(BF p) -> (BF q) -> (BF (Imp p q))
| BF_forall : (p:X->P)((x:X)(BF (p x))) -> (BF (forall p)).

Hint BF_false BF_eq BF_not BF_and BF_imp.

(*-----------------------------------------------------------*)
(*                              RULES                        *)
(*-----------------------------------------------------------*)

(*****************************)
(* Rules of First Order Logic *)
(*****************************)

(* some properties of equality *)
Axiom eq_refl : (t:Te)(w:U)(T w (Eq t t)).
Hint eq_refl.
Axiom eq_congr: (t1,t2:Te)(p:Te->P)(w:U)
(T w (p t1)) ->
(T w (Eq t1 t2)) ->
(BF (p t2)) ->
(T w (p t2)).
Axiom eq_congr_id : (x,y:X)(p:X->P)(w:U)
(T w (p x)) ->
(T w (Eq (isX x) (isX y))) ->
(isnotin x P (forall p)) ->
(isnotin y P (forall p)) ->
(T w (p y)).
```

```
Theorem eq_sym : (t1,t2:Te)(w:U)(T w (Eq t1 t2)) -> (T w (Eq t2 t1)).
Intros t1 t2 w h.
Apply eq_congr with t1:=t1 p:=[t:Te](Eq t t1); Auto.
Qed.
Hint eq_sym.

Theorem eq_trans : (t1,t2,t3:Te)(w:U)
 (T w (Eq t1 t2)) -> (T w (Eq t2 t3)) -> (T w (Eq t1 t3)).
Intros.
Apply eq_congr with t1:=t2 p:=[t:Te](Eq t t3); Auto.
Qed.

(* two delta rules for plus *)
Axiom pls_zero : (t:Te)(w:U)(T w (Eq (pls t zero) t)).
Axiom pls_succ : (t,t1,t2:Te)(w:U)
(T w (Eq (pls t1 t2) t)) ->
(T w (Eq (pls t1 (succ t2)) (succ t))).

Axiom not_E : (p:P)(w:U)(T w p) -> (T w (Not p)) -> (T w false).
Axiom not_I : (p:P)(w:U)((T w p) -> (T w false)) -> (T w (Not p)).

Axiom and_I : (p,q:P)(w:U)(T w p) -> (T w q) -> (T w (And p q)).
Axiom and_El : (p,q:P)(w:U)(T w (And p q)) -> (T w p).
Axiom and_Er : (p,q:P)(w:U)(T w (And p q)) -> (T w q).

Axiom imp_I : (p,q:P)(w:U)((T w p) -> (T w q)) -> (T w (Imp p q)).
Axiom imp_E : (p,q:P)(w:U)(T w (Imp p q)) -> (T w p) -> (T w q).

Axiom forall_I : (p:X -> P)(w:U)
  ((x:X)(isnotin x P (forall p)) -> (T w (p x))) ->
  (T w (forall p)).
Axiom forall_E : (p:X -> P)(q:P)(t:Te)(w:U)
  (T w (forall p)) ->
  ((x:X)(isnotin x Te t) ->
(isnotin x P q) ->
(isnotin x P (forall p)) ->
(T w (Eq (isX x) t)) ->
(T w (p x)) ->
(T w q)) ->
  (T w q).

Axiom notnot_E  : (p:P)(w:U)(T w (Not (Not p))) -> (T w p).

Theorem false_E : (p:P)(w:U)(T w false) -> (T w p).
Intros; Apply notnot_E; Apply not_I; Intro; Assumption.
Qed.
```

```
(***************)
(* Modal Rules *)
(***************)
(* Assignment Rules *)
Axiom ass_I : (p:X -> P)(x:X)(t:Te)(w:U)
      (isnotin x P (forall p)) ->
      ((y:X)(isnotin y X x) ->
    (isnotin y P (forall p)) ->
    (isnotin y Te t) ->
    (T w (Eq (isX y) t)) ->
    (T w (p y)))   ->
      (T w (box (ass x t) (p x))).
Axiom ass_E : (p:X -> P)(q:P)(x:X)(t:Te)(w:U)
      (T w (box (ass x t) (p x))) ->
      ((y:X)(isnotin y Te t) ->
    (isnotin y P q) ->
    (isnotin y P (forall p)) ->
    (T w (Eq (isX y) t)) ->
    (T w (p y)) ->
    (T w q))    ->
      (T w q).

Axiom comp_I : (p:P)(c,d:C)(w:U)
(T w (box c (box d p))) ->
(T w (box (comp c d) p)).
Axiom comp_E : (p:P)(c,d:C)(w:U)
(T w (box (comp c d) p)) ->
(T w (box c (box d p))).

Axiom test_I : (p:P)(b:B)(w:U)
((T w (b2p b)) -> (T w p)) ->
(T w (box (test b) p)).
Axiom test_E : (p:P)(b:B)(w:U)
(T w (box (test b) p)) -> (T w (b2p b)) ->
(T w p).

Axiom ndc_I : (p:P)(c1,c2:C)(w:U)
(T w (box c1 p)) -> (T w (box c2 p)) ->
(T w (box (ndc c1 c2) p)).
Axiom ndc_El : (p:P)(c1,c2:C)(w:U)
(T w (box (ndc c1 c2) p)) ->
(T w (box c1 p)).
Axiom ndc_Er : (p:P)(c1,c2:C)(w:U)
(T w (box (ndc c1 c2) p)) ->
(T w (box c2 p)).
```

```
Fixpoint I [n:nat] : C -> P -> P
 := [c:C][p:P](<P>Case n of
    (*  0  *)   p
    (* S n *) [m:nat](box c (I m c p))
    end).

Axiom iter_I :  (p:P)(c:C)(w:U)
((n:nat)(T w (I n c p))) ->
(T w (box (iter c) p)).
Axiom iter_E :  (p:P)(c:C)(w:U)
(T w (box (iter c) p)) ->
(n:nat)(T w (I n c p)).


(* The base case for Scott rule: only on the assignment *)
Axiom Sc_basecase : (x:X)(t:Te)(p:P)(G:nat->P)
  ((w:U)((n:nat)(T w (G n))) -> (T w p))
  ->
  (w:U) ((n:nat)(T w (box (ass x t) (G n)))) ->
(T w (box (ass x t) p)).

(* Then, it is generalized to every command *)
Theorem Sc : (c:C)(p:P)(G:nat->P)
     ((w:U)((n:nat)(T w (G n))) -> (T w p))
     ->
     (w:U)((n:nat)(T w (box c (G n)))) -> (T w (box c p)).
(Induction c;Intros).
(* case 0: assignment *)
(Apply Sc_basecase with G:=G ;Auto).
(* case 1: composition *)
(Apply comp_I;Apply H with G:=[n:nat](box c1 (G n)) ).
Exact (H0 p G H1).
(Intros;Apply comp_E;Apply H2).
(* case 2: test *)
(Apply test_I;Intro;Apply H;Intro;Apply test_E with b:=b ;Auto).
(* case 3: iteraction *)
(Apply iter_I;Intros n;Cut (n1:nat)(T w (I n c0 (G n1)))).
(Generalize n w ;Induction n). (* by induction on iteration levels *)
   (* base case *)
   Exact H0.
   (* step case *)
   (Intros;Simpl;Apply H with G:=[n:nat](I n1 c0 (G n)) ;Auto).
   (Intro;Apply iter_E;Exact (H1 n1)).
(* case 4: non-deterministic choice *)
Apply ndc_I.
```

```
(Apply H with G:=G ;Auto).
(Intro;Apply ndc_El with c2:=c1 ;Apply H2).
(Apply H0 with G:=G ;Auto).
(Intro;Apply ndc_Er with c1:=c0 ;Apply H2).
Qed.


(* box-intro is a derived rule *)
Theorem box_I : (c:C)(p:P)(w:U)
((w1:U)(T w1 p)) -> (T w (box c p)).
Induction c; Intros.
(* case 0: assignment *)
Apply Sc_basecase with G:=[n:nat](Eq zero zero).
Intros; Apply H.
Intro; Apply ass_I with p:=[x:X](Eq zero zero); Auto.
   Apply isnotin_n with op:=forall;
   Intros; Apply isnotin_2 with op:=Eq; Apply isnotin_zero.
(* case 1 : composition *)
Apply comp_I; Apply H; Intro; Apply H0; Assumption.
(* case 2 : test *)
Apply test_I; Intro; Apply H.
(* case 3 : iteration *)
Apply iter_I; Intro; Generalize n w.
Induction n.
Exact H0.
Intros; Simpl; Apply H; Assumption.
(* case 4 : non deterministic choice *)
Apply ndc_I.
Apply H; Assumption.
Apply H0; Assumption.
Qed.


Theorem K : (c:C)(p,q:P)
    ((w:U)(T w p) -> (T w q))
    ->
    (w:U)(T w (box c p)) -> (T w (box c q)).
Intros; Apply Sc with G:=[n:nat]p; Intros.
Apply H; Exact (H1 O).
Exact H0.
Qed.

(*****************************************)
(*       End of Minimal Dynamic Logic       *)
(*****************************************)
```

```
(*****************************************)
(*        Diamond and derived rules         *)
(*****************************************)

Definition diam := [c:C][p:P](Not (box c (Not p))).
(*
Syntax diam "<_>_".
*)

Theorem contraposition : (p,q:P)(w:U)((T w p) -> (T w q)) ->
 (T w (Not q)) -> (T w (Not p)).
Intros; Apply not_I; Intro; Apply not_E with q; Auto.
Qed.

Theorem notnot_I : (p:P)(w:U)(T w p) -> (T w (Not (Not p))).
Intros; Apply not_I; Intro; Apply not_E with p; Assumption.
Qed.

Theorem K_diam : (c:C)(p,q:P)
  ((w:U)(T w p) -> (T w q))
  ->
  (w:U)(T w (diam c p)) -> (T w (diam c q)).
Unfold diam; Intros; Apply contraposition with q:=(box c (Not p)); Auto.
Intro; Apply K with p:=(Not q); Auto.
Intros; Apply contraposition with q; Auto.
Qed.


Theorem comp_I_diam : (p:P)(c,d:C)
(w:U)(T w (diam c (diam d p))) -> (T w (diam (comp c d) p)).
Unfold diam; Intros;
   Apply contraposition with q:=(box c (Not (Not (box d (Not p))))); Auto.
Intro; Apply K with p:=(box d (Not p)).
Exact (notnot_I (box d (Not p))).
Apply comp_E; Assumption.
Qed.

Theorem comp_E_diam : (p:P)(c,d:C)
(w:U)(T w (diam (comp c d) p)) -> (T w (diam c (diam d p))).
Unfold diam; Intros;
   Apply contraposition with q:=(box (comp c d) (Not p)); Auto.
Intros; Apply comp_I; Apply K with p:=(Not (Not (box d (Not p)))); Auto.
Intros; Apply notnot_E; Auto.
Qed.
```

```
Theorem test_I_diam : (p:P)(b:B)
(w:U)(T w (b2p b)) -> (T w p) -> (T w (diam (test b) p)).
Intros; Unfold diam.
Apply not_I; Intro.
Apply not_E with p.
Assumption.
Apply test_E with b.
Assumption.
Assumption.
Qed.

Theorem test_E1_diam : (p:P)(b:B)
(w:U)(T w (diam (test b) p)) -> (T w (b2p b)).
Unfold diam; Intros; Apply notnot_E;
   Apply contraposition with q:=(box (test b) (Not p)); Auto.
Intros; Apply test_I; Intro; Apply false_E.
Apply not_E with (b2p b); Auto.
Qed.

Theorem test_E2_diam : (p:P)(b:B)
(w:U)(T w (diam (test b) p)) -> (T w p).
Unfold diam; Intros; Apply notnot_E;
   Apply contraposition with q:=(box (test b) (Not p)); Auto.
Intros; Apply test_I; Intro; Assumption.
Qed.

(*******************************************)
(* algebraic properties of regular programs *)
(*******************************************)

(* associativity of composition *)
Theorem Comp_Ass_l : (c,d,e:C)(p:P)(w:U)
  (T w (box (comp c (comp d e)) p)) -> (T w (box (comp (comp c d) e) p)).
Intros; Apply comp_I; Apply comp_I; Apply K with p:=(box (comp d e) p).
Exact (comp_E p d e).
Apply comp_E; Assumption.
Qed.

Theorem Comp_Ass_r : (c,d,e:C)(p:P)(w:U)
  (T w (box (comp (comp c d) e) p)) -> (T w (box (comp c (comp d e)) p)).
Intros; Apply comp_I; Apply K with p:=(box d (box e p)).
Exact (comp_I p d e).
Apply comp_E; Apply comp_E; Assumption.
Qed.

Theorem Comp_Ass_l_diam : (c,d,e:C)(p:P)(w:U)
```

```
  (T w (diam (comp c (comp d e)) p)) -> (T w (diam (comp (comp c d) e) p)).
Unfold diam; Intros; Apply not_I; Intro;
   Apply not_E with (box (comp c (comp d e)) (Not p)); Auto.
Apply Comp_Ass_r; Auto.
Qed.

Theorem Comp_Ass_r_diam : (c,d,e:C)(p:P)(w:U)
  (T w (diam (comp (comp c d) e) p)) -> (T w (diam (comp c (comp d e)) p)).
Unfold diam; Intros; Apply not_I; Intro;
   Apply not_E with (box (comp (comp c d) e) (Not p)); Auto.
Apply Comp_Ass_l; Auto.
Qed.

(* idempotence of * *)
Theorem iter_idem_l : (c:C)(p:P)(w:U)
(T w (box (iter c) p)) ->
(T w (box (comp (iter c) (iter c)) p)).
Intros; Apply comp_I; Apply iter_I; Intro.
Generalize n w H .
Clear  n H w.
(Induction n;Intros;Auto).
Simpl; Apply K with p:=(box (iter c) p).
Assumption.
Apply Sc with G:=[n:nat](I n c p) .
Intros;Apply iter_I;Assumption.
Intro; Replace (box c (I n1 c p)) with (I (S n1) c p).
(Apply iter_E;Assumption).
Trivial.
Qed.

Theorem iter_idem_r : (c:C)(p:P)(w:U)
        (T w (box (comp (iter c) (iter c)) p)) ->
        (T w (box (iter c) p)).
(Intros;Replace (box (iter c) p) with (I O c (box (iter c) p))).
(Apply iter_E;Apply comp_E;Assumption).
Trivial.
Qed.

(*******************************************************)
(*        End of Diamond and some derived rules        *)
(*******************************************************)


(* Convergence Axiom *)
Axiom Conver : (p:Te -> P)(c:C)(t:Te)(w:U)
      ((w1:U)(x:X)
```

```
 (isnotin x P (forall [x:X](p (isX x)))) ->
 (isnotin x C c) ->
 (T w1 (p (succ (isX x)))) ->
(T w1 (diam c (p (isX x))))
) ->
(T w (p t))
->
(T w (diam (iter c) (p zero))).


(* forall-elimination is assignment introduction *)
Theorem feiai : (p:X -> P)(x:X)(t:Te)
(isnotin x P (forall p)) ->
(w:U)(T w (forall p)) -> (T w (box (ass x t) (p x))).
Intros p x t x_new w H; Apply ass_I.
Apply x_new.
Intros; Apply forall_E with p:=p t:=(isX y); Auto; Intros;
   Apply eq_congr_id with x:=x0; Assumption.
Qed.

(* generic assignment is forall introduction *)
Theorem gaifi : (p:X -> P)(x:X)(w:U)
((t:Te)(T w (box (ass x t) (p x)))) -> (T w (forall p)).
Intros; Apply forall_I; Intros x0 H0;
   Apply ass_E with p:=p 1:=(H (isX x0)); Intros;
   Apply eq_congr_id with x:=y y:=x0; Auto.
Qed.

(*********************************************************)
(*         End of Full Dynamic Logic                     *)
(*********************************************************)
```

16.2 Equivalence of while-do and repeat-until

```
(* Equivalence between repeat and while *)
Definition repeat :=
  [c:C][b:B](comp c (comp (iter (comp (test (bNot b)) c)) (test b))).

Theorem While2Repeat : (c:C)(b:B)(p:P)
(w:U)(T w (box (comp c (while (bNot b) c)) p)) ->
     (T w (box (repeat c b) p)).
Unfold while repeat; Intros; Apply Comp_Ass_r; Apply comp_I.
Apply K with p:=(box (test (bNot (bNot b))) p).
Intros; Apply test_I; Intro; Apply test_E with b:=(bNot (bNot b)).
```

```
Assumption.
Simpl; Apply notnot_I; Assumption.
Apply comp_E; Apply Comp_Ass_l; Assumption.
Qed.

Theorem Repeat2While : (c:C)(b:B)(p:P)
(w:U)(T w (box (repeat c b) p)) ->
     (T w (box (comp c (while (bNot b) c)) p)).
Unfold while repeat; Intros; Apply Comp_Ass_r; Apply comp_I.
Apply K with p:=(box (test b) p).
Intros; Apply test_I; Intro; Apply test_E with b:=b.
Assumption.
Simpl in H0; Apply notnot_E; Assumption.
Apply comp_E; Apply Comp_Ass_l; Assumption.
Qed.


Theorem While2Repeat_total : (c:C)(b,b1:B)
(w:U)(T w (diam (comp c (while (bNot b) c)) (b2p b1))) ->
     (T w (diam (repeat c b) (b2p b1))).
Unfold while repeat; Intros; Apply Comp_Ass_r_diam; Apply comp_I_diam.
Apply K_diam with p:=(diam (test (bNot (bNot b))) (b2p b1)).
Intros; Apply test_I_diam.
Cut (T w0 (b2p (bNot (bNot b)))).
Simpl; Intro; Apply notnot_E; Assumption.
Apply test_E1_diam with 1:=H0.
Apply test_E2_diam with 1:=H0.
Apply comp_E_diam; Apply Comp_Ass_l_diam; Assumption.
Qed.

Theorem Repeat2While_total : (c:C)(b,b1:B)
(w:U)(T w (diam (repeat c b) (b2p b1))) ->
     (T w (diam (comp c (while (bNot b) c)) (b2p b1))).
Unfold while repeat; Intros; Apply Comp_Ass_r_diam; Apply comp_I_diam.
Apply K_diam with p:=(diam (test b) (b2p b1)).
Intros; Apply test_I_diam.
Simpl; Apply notnot_I; Apply test_E1_diam with 1:=H0.
Apply test_E2_diam with 1:=H0.
Apply comp_E_diam; Apply Comp_Ass_l_diam; Assumption.
Qed.
```

16.3 Derivation of Hoare Logic

```
Definition HT : U -> P -> C -> P -> Prop
      := [w:U][p:P][c:C][q:P](T w (Imp p (box c q))).


Theorem Ass_Rule : (p:P)(x:X)(t:Te)
(w:U)(HT w (box (ass x t) p) (ass x t) p).
Intros;Unfold HT; Apply imp_I; Intro; Assumption.
Qed.

Theorem If_Rule : (p,q:P)(b:B)(c,d:C)(w:U)
  (HT w (And (b2p b) p) c q) ->
(HT w (And (Not (b2p b)) p) d q)
-> (HT w p (ifte b c d) q).
Unfold HT ifte; Intros.
Apply imp_I; Intro; Apply ndc_I; Apply comp_I; Apply test_I; Simpl; Intro.
Apply imp_E with (And (b2p b) p).
Assumption.
Apply and_I; Assumption.
Apply imp_E with (And (Not (b2p b)) p).
Assumption.
Apply and_I; Assumption.
Qed.

Theorem Comp_Rule : (p,q,r:P)(c,d:C)(w:U)
(HT w p c q) ->
((w1:U)(HT w1 q d r))
-> (HT w p (comp c d) r).
Unfold HT; Intros; Apply imp_I; Intro;
   Apply comp_I; Apply K with p:=q; Intros.
Apply imp_E with p:=q; Auto.
Apply imp_E with p:=p; Assumption.
Qed.

Theorem Cons_Rule : (p,q,r,s:P)(c:C)(w:U)
(T w (Imp p q)) ->
(HT w q c r) ->
((w1:U)(T w1 (Imp r s)))
-> (HT w p c s).
Unfold HT; Intros; Apply imp_I; Intro; Apply K with p:=r; Intros.
Apply imp_E with p:=r; Auto.
Apply imp_E with p:=q; Auto.
Apply imp_E with p:=p; Auto.
Qed.
```

```
Theorem Or_Rule : (p,q:P)(c,d:C)(w:U)
(HT w p c q) ->
(HT w p d q) ->
(HT w p (ndc c d) q).
Unfold HT; Intros; Apply imp_I; Intro; Apply ndc_I;
  Apply imp_E with p:=p; Assumption.
Qed.

Theorem While_Rule : (p:P)(b:B)(c:C)(w:U)
((w1:U)(HT w1 (And p (b2p b)) c p)) ->
        (HT w p (while b c) (And p (Not (b2p b)))).
Intros p b c w InvariantProperty.
Unfold HT while; Apply imp_I; Intro; Apply comp_I; Apply K with p:=p.
Intros; Apply test_I; Simpl; Intro; Apply and_I; Assumption.
Apply iter_I; Intro; Cut (T w p).
Generalize n w; Clear n H w.
Induction n; Intros; Simpl; Auto.
Apply comp_I; Apply test_I; Intro; Apply K with p:=p; Auto.
Cut (HT w (And p (b2p b)) c p).
Unfold HT.
Intro; Apply imp_E with p:=(And p (b2p b)); Auto.
Apply and_I; Auto.
Apply InvariantProperty.
Assumption.
Save.

Theorem WhileTermination_Rule : (p:Te -> P)(b:B)(c:C)
((w:U)(n:X)(T w (Imp (p (succ (isX n))) (b2p b)))) ->
((w:U)(n:X)(T w (Imp (p (succ (isX n))) (diam c (p (isX n)))))) ->
((w:U)(T w (Imp (p zero) (Not (b2p b))))) ->
(w:U)(n:X)(T w (Imp (p (isX n)) (diam (while b c) (p zero)))).
Intros; Unfold while; Apply imp_I; Intro; Apply comp_I_diam.
Apply K_diam with p:=(p zero).
Intros; Apply test_I_diam.
Apply imp_E with p:=(p zero); Auto.
Assumption.
Apply Conver with t:=(isX n); Auto.
Intros; Apply comp_I_diam; Apply test_I_diam;
    Apply imp_E with p:=(p (succ (isX x))); Auto.
Qed.
```

Notice that the internalization in [AHMP92, Section 6.1], differently from the one here presented, encodes the validity consequence relation and not the truth CR.

Chapter 17

The Implementation of μ-calculus

17.1 Implementation of syntax

```
(* Sets for actions, variables *)
Parameter Act : Set.

Parameter var : Set.
(* var is at least enumerable *)
Axiom var_nat : (Ex [srj:var->nat](n:nat)(Ex [x:var](srj x)=n)).

Lemma neverempty : (x:var)(Ex [y:var]~(x=y)).
(Elim var_nat;Intros srj H x;Elim (H (S (srj x)));Intros).
Exists x0.
Unfold not ;Intro.
(Absurd (eq ? (srj x0) (S (srj x)));Auto).
(Rewrite -> H1;Apply n_Sn).
Qed.

(* the set of preformulae, also not well formed *)
Inductive o   : Set :=
  ff  : o
| Not : o -> o
| And : o -> o -> o
| Imp : o -> o -> o
| Box : Act -> o -> o
| Var : var -> o
| mu  : (var->o) -> o.

Fixpoint isin [x:var;A:o] : Prop :=
   <Prop>Case A of
              False
              [B:o](isin x B)
              [A1,A2:o](isin x A1)\/(isin x A2)
```

```
              [A1,A2:o](isin x A1)\/(isin x A2)
              [a:Act][B:o](isin x B)
              [y:var]x=y
              [F:var->o](y:var)(isin x (F y))
         end.

Fixpoint notin [x:var;A:o] : Prop :=
   <Prop>Case A of
              True
              [B:o](notin x B)
              [A1,A2:o](notin x A1)/\(notin x A2)
              [A1,A2:o](notin x A1)/\(notin x A2)
              [a:Act][B:o](notin x B)
              [y:var]~(x=y)
              [F:var->o](y:var)~(x=y) -> (notin x (F y))
         end.

Fixpoint posin [x:var;A:o] : Prop :=
   <Prop>Case A of
      True
      [B:o](negin x B)
      [A1,A2:o](posin x A1)/\(posin x A2)
      [A1,A2:o](negin x A1)/\(posin x A2)
      [a:Act][A1:o](posin x A1)
      [y:var]True
      [F:var->o](y:var)~(x=y) -> (posin x (F y))
 end
with negin [x:var;A:o] : Prop :=
   <Prop>Case A of
              True
              [B:o](posin x B)
              [A1,A2:o](negin x A1)/\(negin x A2)
              [A1,A2:o](posin x A1)/\(negin x A2)
              [a:Act][A1:o](negin x A1)
              [y:var]~(x=y)
              [F:var->o](y:var)~(x=y) -> (negin x (F y))
         end.

Fixpoint iswf [A:o] : Prop :=
   <Prop>Case A of
      True
      [A1:o](iswf A1)
      [A1:o][A2:o](iswf A1)/\(iswf A2)
      [A1:o][A2:o](iswf A1)/\(iswf A2)
      [a:Act][A1:o](iswf A1)
      [x:var]True
```

```
      [F:var->o](x:var)
      ((notin x (mu F)) -> (posin x (F x)))/\(iswf (F x))
 end.

(* the set of well formed formuale *)
Record wfo : Set := mkwfo {
     prp : o;
     cnd : (iswf prp)
     }.

(* now some results, relating isin, notin, posin, negin *)

(* separation: if x does not apper in A and y do, then x and y are
 * not the same identifiers *)
Lemma separation : (x,y:var)(A:o)(notin x A) -> (isin y A) -> ~(x=y).
(Induction A;Intros;Simpl in H;Simpl in H0).
(* case ff *)
Contradiction.
(* case not *)
(Apply H;Assumption).
(* case and *)
(Elim H1;Elim H2;Intros).
(Apply H;Assumption).
(Apply H0;Assumption).
(* case imp *)
(Elim H1;Elim H2;Intros).
(Apply H;Assumption).
(Apply H0;Assumption).
(* case box *)
(Apply H;Assumption).
(* case var *)
(Rewrite H0;Assumption).
(* case mu *)
(Elim (neverempty x);Intros;Apply (H x0)).
Apply H0; Assumption.
Apply H1.
Qed.

(* an identifier which does not occurr,
 * occurrs both positively and negatively *)

Lemma notin_posin_negin :
         (x:var)(A:o)(notin x A) -> (posin x A)/\(negin x A).
(Induction A;Intros; Auto).
(Elim (H H0);Intros;Split;Assumption).
(Elim H1;Intros; Elim (H H2);Elim (H0 H3);Intros;Split;Simpl;Auto).
```

```
(Elim H1;Intros; Elim (H H2);Elim (H0 H3);Intros;Split;Simpl;Auto).
(Simpl;Auto).
(Simpl; Split; Intros; Elim (H y (H0 y H1)); Intros; Assumption).
Qed.

Lemma notin_posin : (x:var)(A:o)(notin x A) -> (posin x A).
(Intros; Apply proj1 with B:=(negin x A);
 Apply notin_posin_negin; Assumption).
Qed.

Lemma notin_negin : (x:var)(A:o)(notin x A) -> (negin x A).
(Intros; Apply proj2 with A:=(posin x A);
 Apply notin_posin_negin; Assumption).
Qed.


(* And now, as example, a formula in which we need the separation
lemma for proving its well-formness: (mu z.~x) *)
Variable x:var.

Lemma f1:wfo.
Apply (mkwfo (mu [z:var](Not (Var x)))).
(Simpl;Split;Auto).
(Intros;Apply separation with A:=(mu [z:var](Not (Var x))) ;Simpl;Auto).
Qed.
Transparent f1.

(* The "transparent" is needed in order to make unfoldable the pair
  <prp,cnd> denoted by f1 - by default, lemmata are not transparent
  (does not delta-reduce).
*)

(********************)
(* the substitution *)
(********************)

Inductive subst [A:o] : (var->o) -> o -> Prop :=
    subst_ff   : (subst A [x:var]ff ff)
  | subst_not  : (B:var->o)(C:o)
        (subst A B C) ->
 (subst A [x:var](Not (B x)) (Not C))
         | subst_and  : (B1,B2:var->o)(C1,C2:o)
                        (subst A B1 C1) -> (subst A B2 C2) ->
 (subst A [x:var](And (B1 x) (B2 x)) (And C1 C2))
         | subst_imp  : (B1,B2:var->o)(C1,C2:o)
                        (subst A B1 C1) -> (subst A B2 C2) ->
```

```
                          (subst A [x:var](Imp (B1 x) (B2 x)) (Imp C1 C2))
          | subst_box  : (a:Act)(B:var->o)(C:o)
                          (subst A B C) ->
 (subst A [x:var](Box a (B x)) (Box a C))
          | subst_xx   : (subst A [x:var](Var x) A)
  | subst_xy   : (y:var)(subst A [x:var](Var y) (Var y))
  | subst_mu   : (B:var->var->o)(C:var->o)
 ((y:var)(subst A [x:var](B x y) (C y))) ->
 (subst A [x:var](mu (B x)) (mu C)).
Hint subst_ff subst_not subst_and subst_imp
    subst_box subst_xx subst_xy subst_mu.

(* a vacuous substitution has no effect *)
Lemma subst_not_free : (A,B:o)(subst A [x:var]B B).
Induction B; Intros.
Apply subst_ff.
Apply subst_not with B:=[x:var]o0; Assumption.
Apply subst_and with B1:=[x:var]o0 B2:=[x:var]o1; Assumption.
Apply subst_imp with B1:=[x:var]o0 B2:=[x:var]o1; Assumption.
Apply subst_box with B:=[x:var]o0; Assumption.
Apply subst_xy.
Apply subst_mu with B:=[x:var]o0; Assumption.
Qed.


(* example: (mu y.(x/\y))[x:=ff] = (mu y.(ff/\y)) *)

Goal (subst ff [x:var](mu [y:var](And (Var x) (Var y)))
               (mu [y:var](And ff (Var y)))).
Apply subst_mu with B:=[x,y:var](And (Var x) (Var y)); Intro.
Apply subst_and with B1:=[x:var](Var x) B2:=[x:var](Var y) .
Apply subst_xx.
Apply subst_xy.
Qed.
```

17.2 Implementation of proof system

```
(* the universe, for the world technique *)
Parameter U:Set.

(* the proving judgement *)
Parameter T : U -> o -> Prop.

Section Proof_Rules.
Variable A,B:o.
```

```
Variable w:U.

(* proof rules operate also on non-well formed formulae, but for
having the soundness of the system, we need to require well-formness
of every discharged formula *)

Axiom ff_I   : (T w A) -> (T w (Not A)) -> (T w ff).
Axiom Not_I  : (iswf A) -> ((T w A) -> (T w ff)) -> (T w (Not A)).
Axiom RAA    : (iswf A) -> ((T w (Not A)) -> (T w ff)) -> (T w A).

Axiom And_I  : (T w A) -> (T w B) -> (T w (And A B)).
Axiom And_El : (T w (And A B)) -> (T w A).
Axiom And_Er : (T w (And A B)) -> (T w B).

Axiom Imp_I  : (iswf A) -> ((T w A) -> (T w B)) -> (T w (Imp A B)).
Axiom Imp_E  : (T w (Imp A B)) -> (T w A) -> (T w B).

Axiom Scott  : (G:nat->o)(a:Act)
       ((w':U)((n:nat)(T w' (G n))) -> (T w' A))
       ->
       ((n:nat)(T w (Box a (G n)))) -> (T w (Box a A)).

Axiom mu_I   : (F:var->o)
                ((z:var)(notin z (mu F)) -> (Var z)=(mu F) -> (T w (F z)))
       -> (T w (mu F)).
Axiom mu_E   : (F:var->o)(iswf A) ->
       ((z:var)(notin z (mu F)) -> (Var z)=A ->
                       (w':U)(T w' (F z)) -> (T w' A))
       ->
       (T w (mu F)) -> (T w A).

End Proof_Rules.

Lemma ff_E : (A:o)(iswf A) -> (w:U)(T w ff) -> (T w A).
Intros; Apply RAA; Intros; Assumption.
Qed.

Lemma K : (A,B:o)(a:Act)(w:U)
  ((w':U)(T w' A) -> (T w' B))
  ->
  (T w (Box a A)) -> (T w (Box a B)).
Intros;Apply Scott with G:=[n:nat]A; Intros.
Apply H; Exact (H1 O).
Exact H0.
Qed.
```

17.3 An example of derivation

```
(* An example *)
(* A <-> mu x.((not A) => x) *)
Lemma ex1 : (A:wfo)(w:U)
        (T w (prp A)) <-> (T w (mu [x:var](Imp (Not (prp A)) (Var x)))).
(Intros;Split;Intro).

(* -> *)
Apply mu_I; Intros; Apply Imp_I;Intros.
(* Here we have the first test of well formness - but this is easy *)
Exact (cnd A).
(* Then we can go on *)
Rewrite H1. (* this is the key of the substution *)
Apply ff_E.
(* Here we are with the second test *)
Split.
(Intro;Split).
(* now we are facing a huge and painful term - but don't worry *)
Apply notin_posin.
(Elim (neverempty x);Intros).
(Apply proj1 with B:=(not (eq ? x x0)) ;Simpl in H3;Apply H3;Assumption).
(Simpl; Trivial).
Split.
Exact (cnd A).
Simpl; Trivial.
(* oh well, now we can go on *)
(Apply ff_I with A:=(prp A) ;Assumption).

(* <- : the goal is as follows:
  A : wfo
  w : U
  H : (T w (mu [x:var](Imp (Not (prp A)) (Var x))))
  ============================
   (T w (prp A))
*)
Apply mu_E with F:=[x:var](Imp (Not (prp A)) (Var x)); Intros.

(* now we have the third test - it is easy *)
Exact (cnd A). (* and it's over *)
(* and now we can go on -
   the first goal does not depend on w and assumption H any more:
  A : wfo
  w : U
  H : (T w (mu [x:var](Imp (Not (prp A)) (Var x))))
  z : var
```

```
  H0 : (notin z (mu [x:var](Imp (Not (prp A)) (Var x))))
  H1 : (Var z)=(prp A)
  w' : U
  H2 : (T w' (Imp (Not (prp A)) (Var z)))
  ============================
   (T w' (prp A))
so we can drop them *)
Clear  H w.
(* now we have completely changed the sequent:
  A : wfo
  z : var
  H0 : (notin z (mu [x:var](Imp (Not (prp A)) (Var x))))
  H1 : (Var z)=(prp A)
  w' : U
  H2 : (T w' (Imp (Not (prp A)) (Var z)))
  ============================
   (T w' (prp A))
*)
Apply RAA;Intros.
(* now we have the last check *)
Exact (cnd A).
(* and now we can complete the proof *)
Apply ff_I with A:=(prp A).
Rewrite <- H1.
Apply Imp_E with A:=(Not (prp A)); Assumption.
Assumption.
Assumption.
Qed.
```

Chapter 18

Conclusions and Future Work

In this thesis, we have investigated the formalization of logical theories of languages and programs, in Logical Frameworks based on Type Theories.

Although very successful in dealing with "purely logical" systems, the "standard" representation paradigm of Logical Framework does not apply satisfactorly to formal systems for proving properties of programs. These systems often present many idiosyncrasies which escape the standard paradigm, such as negative formulæ constructors, non-standard notions of instantiation and substitution, context-sensitive grammars, typing systems, infinitary formulæ, subsorting, equivalence theories of expressions, infinitary rules, impure rules, and so on.

The solution proposed in this thesis is that, in order to get the best result, one has to take into account the characteristics both of the Framework (meta system) and of the object system. As far as Logical Frameworks are concerned, we need to develop and investigate new efficient representation techniques, taking as much advantage as possible of the structural features of the metalanguage. Moreover, the metalogical features of Logical Frameworks have also a retrospective effect on the design of new representations of the formal systems themselves. This reformulation process could involve both the syntactic and the deductive part of the formal system. Often, the conceptual understanding yielded by the encoding methodology suggests new systems in Natural Deduction style, which simplify both the encoding process and the subsequent usage.

The proposed methodology has been described and tested through the presentation of a rich variety of logical systems: Structured and Natural Operational Semantics, Modal Logics, Dynamic Logics, μ-calculus. Each of these paradigmatical examples presents distinctive features whose formal representation in Logical Framework is problematic. In each case, we have proposed, discussed and proved correct, one or several solutions; in this venture, we have faced many issues regarding both the metatheory and the formal representation, in Logical Frameworks, of these logical systems. In many cases, genuinely new formal systems, in Natural Deduction style, have been introduced. At the metalogical level, we generalize and combine the concept of consequence relation introduced by Avron and Aczel, in order to handle *schematic* and *multiple* consequences.

We focused on a particular Logical Frameworks, namely the *Calculus of Inductive Constructions* originated by Coquand and Huet, and its implementation, Coq, developed at the INRIA and the ENS-Lyon. The investigation carried out in this thesis shows the wide applicability of this framework, following the proposed methodology. However, this

research has pointed out also some limitations of existing Logical Frameworks, in relation with the encoding of program logics; our analysis could be therefore the starting point for further improvements of these metalanguages.

This work would serve as a guide and reference to future users of Logical Frameworks.

18.1 A more detailed overview

Although it is difficult to convey the great deal of details, techniques, problems and solutions which have arisen along this thesis, we briefly recall the main results.

On the metalogical level, we have focused on the notion of Consequence Relation as semantic counterpart of logical systems (Chapter 3). In order to capture correctly the proof systems presented in this thesis, we have generalized and combined the notions of simple consequence relation introduced by Avron and Aczel. We care both of *schematic* (i.e. closed under complex notions of substitution) and of *multiple* (i.e. describing several judgements at once) consequences, in a Natural Deduction setting.

On the logical level, we have investigated *Natural Deduction* style presentations for many logical theories of programs: Structured and Natural Operational Semantics (Chapter 4), Modal Logics (Chapter 5), Propositional and First-Order Dynamic Logic (Chapters 6, 7), Hoare Logic (chapter 8) and the μ-calculus (Chapter 9). This made us to face several subtleties, peculiar to program logics, such as complex notions of substitutions, clashing between logical variables and program variables, recursive formulæ constructors, context-sensitive grammars, modalities, infinitary rules.

We tried to take full advantage of the Natural Deduction style, in order to simplify the subsequent encodings and usage. This metalogical analysis has suggested new presentations of object logics, employing new general techniques based on features of the metalanguage (such as the *bookkeeping technique* for representing environmental informations, or infinitary systems for Dynamic Logics). We have proved the completeness of these systems, with respect to the corresponding consequence relation. We have also addressed possible uses (e.g., proving equivalence of programs), and metatheoretic properties (such as proof-theory) of these new proof systems

On the level of formal representation, we have investigated the encoding of the systems previously introduced, in type-theory based Logical Frameworks, such as the Edinburgh LF and the CIC. In this venture, we faced the representation of problematic features of the object logics, such as binding operators (α-equivalence) with non-standard notions of substitutions, context-sensitive grammars, modal and infinitary rules (Part III). We studied when and how much of these aspects can be delegated to the metalanguage features.

We have investigated the trade-offs between *first-order* and *higher-order* abstract syntax approaches, in relation with inductive definitions. Moreover, we have proposed a slight generalization of CIC inductive definitions which would allow to capture faithfully and naturally a larger class of languages (e.g., context-sensitive grammars) (Chapter 11).

Issue	Theory	LF	Reference(s)
Polyadic operators	UNITY	CIC	[Add94]
	π-calculus	CIC	[Sca97]
Polymorphism	Damas-Milner calculus	ELF	[Har90]
Reflection principle	PTS theory	ECC	[MP93]
Explicit substitutions	$\lambda\sigma$	CIC	[Saï96]
Coinductive datatypes	CCS, process algebræ	CIC	[Fel96, Gim95]

Figure 18.1: Some other complex issues not faced in this thesis.

We have presented a methodology for encoding general transition systems, specified by means of Structural or Natural Operational Semantics. In particular, we have described the implementation of the bookkeeping technique, previously introduced (Chapter 12).

Furthermore, we have introduced and experimented several new techniques for encoding modal and infinitary rules (Chapters 13, 14)

At the pragmatical level, we have experimented with the proposed encoding, by implementing the encodings of Propositional and First-Order Dynamic Logics, and of the μ-calculus, in the Coq system (Chapters 16, 17). Within these implementations, we have carried out some experiments, such as the formal derivation of Hoare Logic in *DL*, and the proof of some simple program equivalence.

In these implementations, we have taken advantage of the pragmatic features provided by Coq, such as automated resolution and conversion tactics. This work has pointed out the flexibility and suitability of the Coq proof environment, in reasoning about properties of programs.

18.2 Future Work

There are so many program logics, and so many formal systems for reasoning on program behaviours, that probably it is impossible to capture every problem one may encounter in view of the specification of a logical theory of programs. In this section we outline some directions for further developments of the subject of this thesis.

18.2.1 Other systems and problematic issues

There are still many problematic issues on program logics, which are not discussed in this thesis. In Figure 18.1 we give a (non-exhaustive) list of some references for some of these problems.

Polyadic bindings are common in many process algebræ, such as the polyadic π-calculus, the polyadic CCS and the UNITY formalism. Their implementation is not immediate, since the metalanguage features only a monadic binding operator (the λ).

Moreover, typing systems of real languages are usually much more complex than PTS's. We have already seen that subsorting is poorly handled by Logical Framework (Section 11.4.4). Other complex features are *polymorphic* and *overloaded* types.

Another interesting toping which deserves further investigations is the representation of infinite or circular objects, via coinductive definitions. In fact, the CIC provides also

coinductive definitions, which can be fruitfully used in representing "infinite terms" such as *streams* or *infinite computations*.

18.2.2 Proof theory

The introduction of genuinely new proof systems, in Natural Deduction style, for program logics (such as, e.g., those for Dynamic Logics, or the μ-calculus) raises many proof-theoretic issues. Although not strictly related to our work, an investigation of proof-theoretic properties of the Natural Deduction systems could yield quite interesting syntactic proofs of significant properties.

18.2.3 Case studies

Many interesting case studies can be carried out on the logics presented and encoded in this thesis. For instance, one can prove formally the correctness of simple programs by using the encoding of Dynamic Logics.

A more interesting case study could be proving the correctness of an interpreter, by means of Dynamic Logic. This can be achieved in three steps:

1. give the NOS semantics for a functional language with regular programs; for instance, the evaluation of an expression M after executing a program c is represented by a proof of $\vdash (\mathbf{do}\ c\ \mathbf{result}\ M) \Downarrow m$.

2. give the Dynamic Logic, where first-order terms are expressions.

3. prove formally its correctness, by proving a statement similar to the following

   Conjecture 18.1 $\vdash \mathbf{do}\ c\ \mathbf{result}\ M \Downarrow m \ \Rightarrow\ \vdash [c](M = m)$

   This property should be provable by induction on the proof of $\vdash \mathbf{do}\ c\ \mathbf{result}\ M \Downarrow m$; hence, the NOS should be given by using an Inductive definition.

18.2.4 Front-end interfaces

In carrying out derivations in our encoding, we have experienced the limited interactivity offered by the raw Coq implementation. Therefore, a user-friendly front-end interface is highly advisable. An interesting and flexible front-end interface for Coq, the CtCoq, has been developed by the CROAP group at the INRIA-Sophia Antipolis [BB96]. An interesting research should be the investigation of how CtCoq can be fruitfully employed in proving properties of the systems presented in this thesis.

18.2.5 Denotational Semantics

In this thesis, we have not faced the formal representation and implementation of denotational specifications. There are two main lines of research about the encoding of Denotational Semantics.

The first approach (*classical*) is to encode directly the theory of cpo's, and reason formally about it. There are several works along this direction, starting from the LCF

experience. This work has begun, at some extent, by Gilles Kahn, in order to to certify the paper [KP93a]; with the help of Gérard Huet, Kahn has completed the development of part of set theory, and has since begun a development of group theory. Aczel encoded Category Theory in Coq. A different work has been done by Cenciarelli, in encoding Moggi's Modular Semantics in LEGO [Cen94].

The second approach (*constructive*) is to give a logical and finitary presentation to domains, and implement the resulting logic. This is possible by means of *Scott's informations systems* [Sco82, Abr87, EHR92, CDHL82], but usually the resulting system is not natural nor user-friendly.

Appendix A

Semantics of $\mathcal{L}_P$, $\mathcal{L}_D$, $\mathcal{L}_{M_F}$, and $\mathcal{L}_{M_I}$

A.1 Rules for the NOS

A.1.1 NOS of $\mathcal{L}_P$

Rules for judgement $\Rightarrow$

For the meaning of $\mathsf{EC}(\cdot,\cdot,\cdot)$, see Section 4.2.2. For typographical reasons, **lambda** will be sometimes abbreviated with $\boldsymbol{\lambda}$ and $[\langle\rangle|C]N$ with $[C]N$.

$$\frac{\begin{array}{c}(x' \Rightarrow n)\\ \vdots\\ \mathit{value}\ n\ \ M' \Rightarrow m'\end{array}}{[n/x]M \Rightarrow m}\ \mathsf{EC}(x,M,m) \tag{A.1}$$

$$\frac{\mathit{value}\ m}{m \Rightarrow m} \tag{A.2}$$

$$\frac{N \Rightarrow n \quad [n/x]M \Rightarrow m}{\mathbf{let}\ x = N\ \mathbf{in}\ M \Rightarrow m} \tag{A.3}$$

$$\frac{\begin{array}{c}(\mathit{closed}\ x)\\ \vdots\\ \mathit{closed}\ M\end{array}}{\mathbf{lambda}\ x.M \Rightarrow \mathbf{lambda}\ x.M} \tag{A.4}$$

$$\frac{\begin{array}{c}(\mathit{closed}\ y)\\ \vdots\\ y \Rightarrow n \quad \mathbf{lambda}\ x.M \Rightarrow m\end{array}}{\mathbf{lambda}\ x.M \Rightarrow [n/y]m} \tag{A.5}$$

$$\frac{M \Rightarrow m \quad N \Rightarrow n \quad m \cdot n \Rightarrow p}{MN \Rightarrow p} \tag{A.6}$$

$$\frac{[n/x]M \Rightarrow p}{(\mathbf{lambda}\ x.M) \cdot n \Rightarrow p} \tag{A.7}$$

$$\frac{\mathit{value}\ n \quad [m'/x](m \cdot n) \Rightarrow p}{([m'/x]m) \cdot n \Rightarrow p} \tag{A.8}$$

$$\frac{\mathit{value}\ n}{(\mathit{plus} \cdot 0) \cdot n \Rightarrow n} \tag{A.9}$$

$$\frac{(\mathit{plus} \cdot m) \cdot n \Rightarrow p}{(\mathit{plus} \cdot (\mathit{succ} \cdot m)) \cdot n \Rightarrow \mathit{succ} \cdot p} \tag{A.10}$$

$$\frac{\begin{array}{l}\mathbf{let} f = (\mathbf{lambda} x.\mathbf{letrec}\\ \qquad f(x) = N\ \mathbf{in} N)\ \mathbf{in} M \Rightarrow p\end{array}}{\mathbf{letrec}\ f(x) = N\ \mathbf{in}\ M \Rightarrow p} \tag{A.11}$$

$$\frac{M \Rightarrow m \quad N \Rightarrow n}{M :: N \Rightarrow m :: n} \tag{A.12}$$

$$\frac{\mathit{value}\ m}{\mathit{hd} \cdot (m :: n) \Rightarrow m} \tag{A.13}$$

$$\frac{\mathit{value}\ n}{\mathit{tl} \cdot (m :: n) \Rightarrow n} \tag{A.14}$$

$$\frac{D \rhd I \quad \mathit{free}\ C\ I \quad [D|C]N \Rightarrow n}{[\mathbf{on}\ D\ \mathbf{do}\ C]N \Rightarrow n} \tag{A.15}$$

$$\frac{M \Rightarrow m \quad [m/x]([D|C]N) \Rightarrow n}{[x = M; D|C]N \Rightarrow n} \tag{A.16}$$

$$\frac{N \Rightarrow n \quad [n/x]M \Rightarrow m}{[x := N]M \Rightarrow m} \tag{A.17}$$

$$\frac{[C]([D]M) \Rightarrow m}{[C; D]M \Rightarrow m} \tag{A.18}$$

$$\frac{M \Rightarrow \mathit{true} \quad [C]N \Rightarrow n}{[\textbf{if } M \textbf{ then } C \textbf{ else } D]N \Rightarrow n} \tag{A.19}$$

$$\frac{M \Rightarrow \mathit{false} \quad [D]N \Rightarrow n}{[\textbf{if } M \textbf{ then } C \textbf{ else } D]N \Rightarrow n} \tag{A.20}$$

$$\frac{\begin{array}{c} M \Rightarrow \mathit{true} \\ [C]([\textbf{while } M \textbf{ do } C]N) \Rightarrow n \end{array}}{[\textbf{while } M \textbf{ do } C]N \Rightarrow n} \tag{A.21}$$

$$\frac{M \Rightarrow \mathit{false} \quad N \Rightarrow n}{[\textbf{while } M \textbf{ do } C]N \Rightarrow n} \tag{A.22}$$

$$\frac{N \Rightarrow n \quad [[n/x]_c C]M \Rightarrow m}{[\textbf{begin new } x = N;\ C \textbf{ end}]M \Rightarrow m} \tag{A.23}$$

$$\frac{\begin{array}{c} (x' \Rightarrow n) \\ \vdots \\ \mathit{value}\ n \quad [C']M \Rightarrow m' \end{array}}{[[n/x]_c C]M \Rightarrow m}\ \mathsf{EC}(x, C, m) \tag{A.24}$$

$$\frac{\mathit{value}\ n}{(\leq \cdot 0) \cdot n \Rightarrow \mathit{true}} \tag{A.25}$$

$$\frac{\mathit{value}\ n}{(\leq \cdot(\mathit{succ} \cdot n)) \cdot 0 \Rightarrow \mathit{false}} \tag{A.26}$$

$$\frac{(\leq \cdot n) \cdot m \Rightarrow p}{(\leq \cdot(\mathit{succ} \cdot n)) \cdot (\mathit{succ} \cdot m) \Rightarrow p} \tag{A.27}$$

$$\frac{[[\textbf{lambda } x, y.C/P]_{pc} D]M \Rightarrow m}{[\textbf{procedure } P(x, y) = C \textbf{ in } D]M \Rightarrow m} \tag{A.28}$$

$$\frac{\begin{array}{c} (P' \Rightarrow_p \boldsymbol{\lambda} x, y.C) \\ \vdots \\ \mathit{free}_c\ C\ (x, y) \quad [D']M \Rightarrow m' \end{array}}{[[\boldsymbol{\lambda} x, y.C/P]_{pc} D]M \Rightarrow m}\ \mathsf{EC}(P, D, m) \tag{A.29}$$

$$\frac{\begin{array}{c} P \Rightarrow \boldsymbol{\lambda} x, y.C \quad M \Rightarrow m \quad z \Rightarrow p \\ [p/x][m/y][C]x \Rightarrow v \quad [v/z]N \Rightarrow n \end{array}}{[P(z, M)]N \Rightarrow n} \tag{A.30}$$

$$\frac{\begin{array}{c} (\mathit{closed}\ P) \\ \vdots \\ \mathit{free}\ C\ (x, y) \quad P \Rightarrow_p \boldsymbol{\lambda} x, y.C \end{array} \quad \boldsymbol{\lambda} z.M \Rightarrow m}{\boldsymbol{\lambda} z.M \Rightarrow [\boldsymbol{\lambda} x, y.C/P]_{pe} m} \tag{A.31}$$

$$\frac{\mathit{value}\ n \quad [Q/P]_{pe}(m \cdot n) \Rightarrow p}{([Q/P]_{pe} m) \cdot n \Rightarrow p} \tag{A.32}$$

$$\frac{\begin{array}{c} (P' \Rightarrow_p Q) \\ \vdots \\ \mathit{free}_c\ C\ (x, y) \quad M' \Rightarrow m' \end{array}}{[\boldsymbol{\lambda} x, y.C/P]_{pe} M \Rightarrow m}\ \mathsf{EC}(P, M, m) \tag{A.33}$$

Rules for judgement *value*

$$\frac{}{\mathit{value}\ m}\ m \text{ is a constant} \tag{A.34}$$

$$\frac{M \Rightarrow m}{\mathit{value}\ m} \tag{A.35}$$

Rules for judgement $\triangleright$

$$\frac{}{\langle\rangle \triangleright \mathit{nil}} \tag{A.36}$$

$$\frac{D_l \triangleright I}{x = M; D_l \triangleright x :: I} \tag{A.37}$$

Rules for judgement *closed*

$$\frac{}{\mathit{closed}\ m}\ m \text{ is a constant} \tag{A.38}$$

$$\frac{\mathit{closed}\ M \quad \mathit{closed}\ N}{\mathit{closed}(MN)} \tag{A.39}$$

$$\frac{\mathit{closed}\ m \quad \mathit{closed}\ n}{\mathit{closed}(m \cdot n)} \tag{A.40}$$

$$\frac{\begin{array}{c} (\mathit{closed}\ x) \\ \vdots \\ \mathit{closed}\ N \quad \mathit{closed}\ M \end{array}}{\mathit{closed}(\textbf{let } x = N \textbf{ in } M)} \tag{A.41}$$

$$\frac{\begin{array}{cc} (\mathit{closed}\ f, \mathit{closed}\ x) & (\mathit{closed}\ f) \\ \vdots & \vdots \\ \mathit{closed}\ N & \mathit{closed}\ M \end{array}}{\mathit{closed}(\textbf{letrec } f(x) = N \textbf{ in } M)} \tag{A.42}$$

$$\frac{\begin{array}{c} (\mathit{closed}\ x) \\ \vdots \\ \mathit{closed}\ M \end{array}}{\mathit{closed}(\textbf{lambda } x.M)} \tag{A.43}$$

$$\frac{\begin{array}{c} (\mathit{closed}\ x) \\ \vdots \\ \mathit{closed}\ n \quad \mathit{closed}\ M \end{array}}{\mathit{closed}([n/x]M)} \tag{A.44}$$

$$\frac{\mathit{closed}\ m \quad \mathit{closed}\ n}{\mathit{closed}(m :: n)} \tag{A.45}$$

$$\frac{D \rhd I \quad \mathit{free}\ C\ I \quad \mathit{closed}\ [D|\mathbf{nop}]M}{\mathit{closed}\ [\mathbf{on}\ D\ \mathbf{do}\ C]M} \tag{A.46}$$

$$\frac{\mathit{closed}\ M}{\mathit{closed}\ [\langle\rangle|\mathbf{nop}]M} \tag{A.47}$$

$$\frac{\begin{array}{c}(\mathit{closed}\ x)\\ \vdots\end{array} \quad \begin{array}{c}\mathit{closed}\ N \quad \mathit{closed}\ [R|\mathbf{nop}]M\end{array}}{\mathit{closed}\ [x = N; R|\mathbf{nop}]M} \tag{A.48}$$

$$\frac{\begin{array}{c}(\mathit{closed}_p\ P)\\ \vdots\end{array} \quad \begin{array}{c}\mathit{free}\ C\ (x, y) \quad \mathit{closed}\ M\end{array}}{\mathit{closed}\ [\mathbf{lambda}x, y.C/P]_{pe}M} \tag{A.49}$$

Rules for judgement *free*

$$\frac{}{\mathit{free}\ x\ (x, m)} \tag{A.50}$$

$$\frac{\mathit{free}\ x\ m}{\mathit{free}\ x\ (y, m)} \tag{A.51}$$

$$\frac{\mathit{free}\ M\ m \quad \mathit{free}\ N\ m}{\mathit{free}\ (M\ N)\ m} \tag{A.52}$$

$$\frac{\mathit{free}\ M\ m \quad \mathit{free}\ N\ x :: m}{\mathit{free}\ (\mathbf{let}x = M\ \mathbf{in}N)\ m} \tag{A.53}$$

$$\frac{\mathit{free}\ M\ x :: m}{\mathit{free}\ (\mathbf{lambda}x.M)\ m} \tag{A.54}$$

$$\frac{\mathit{free}\ M\ m \quad \mathit{free}\ N\ m}{\mathit{free}\ (M \cdot N)\ m} \tag{A.55}$$

$$\frac{\mathit{free}\ M\ m \quad \mathit{free}\ N\ m}{\mathit{free}\ (M :: N)\ m} \tag{A.56}$$

$$\frac{\mathit{free}\ n\ m \quad \mathit{free}\ M\ x :: m}{\mathit{free}\ ([n/x]M)\ m} \tag{A.57}$$

$$\frac{\begin{array}{c}(\mathit{closed}_p\ P)\\ \vdots\end{array} \quad \begin{array}{c}\mathit{free}\ C\ (x, y, m) \quad \mathit{free}\ M\ m\end{array}}{\mathit{free}\ ([\mathbf{lambda}x, y.C/P]M)\ m} \tag{A.58}$$

$$\frac{\mathit{free}\ C\ m \quad \mathit{free}\ M\ m}{\mathit{free}\ ([C]M)\ m} \tag{A.59}$$

$$\frac{\mathit{free}\ C\ m \quad \mathit{free}\ D\ m}{\mathit{free}\ (C; D)\ m} \tag{A.60}$$

$$\frac{\mathit{free}\ M\ m \quad \mathit{free}\ C\ m \quad \mathit{free}\ D\ m}{\mathit{free}\ (\mathbf{if}M\ \mathbf{then}C\ \mathbf{else}D)\ m} \tag{A.61}$$

$$\frac{\mathit{free}\ M\ m \quad \mathit{free}\ C\ m}{\mathit{free}\ (\mathbf{while}M\ \mathbf{do}C)\ m} \tag{A.62}$$

$$\frac{\mathit{free}\ M\ m \quad \mathit{free}\ C\ x :: m}{\mathit{free}\ (\mathbf{begin}\ \mathbf{new}x = M; C\ \mathbf{end})\ m} \tag{A.63}$$

$$\frac{\begin{array}{c}(\mathit{closed}_p\ P)\\ \vdots\end{array} \quad \begin{array}{c}\mathit{free}\ C\ (x, y, m) \quad \mathit{free}\ D\ m\end{array}}{\mathit{free}\ (\mathbf{proc}P(x, y) = C\ \mathbf{in}D)\ m} \tag{A.64}$$

$$\frac{\mathit{closed}_p(P) \quad \mathit{free}\ x\ m \quad \mathit{free}\ M\ m}{\mathit{free}\ (P(x, M))\ m} \tag{A.65}$$

$$\frac{\mathit{free}\ n\ m \quad \mathit{free}\ C\ x :: m}{\mathit{free}\ ([n/x]C)\ m} \tag{A.66}$$

$$\frac{\begin{array}{c}(\mathit{closed}_p\ P)\\ \vdots\end{array} \quad \begin{array}{c}\mathit{free}\ C\ (x, y, m) \quad \mathit{free}\ D\ m\end{array}}{\mathit{free}\ ([\mathbf{lambda}x, y.C/P]D)\ m} \tag{A.67}$$

A.1.2 NOS of $\mathcal{L}_D$

Rules for judgement $\Rightarrow$

$$\frac{R \Rightarrow_d r \quad \{r\}M \Rightarrow m}{\mathbf{let}\ R\ \mathbf{in}\ M \Rightarrow m} \tag{A.68}$$

$$\frac{M \Rightarrow m}{\{nil\}M \Rightarrow m} \tag{A.69}$$

$$\frac{[n/x]M \Rightarrow m}{\{x \mapsto n\}M \Rightarrow m} \tag{A.70}$$

$$\frac{\{r\}(\{s\}M) \Rightarrow m}{\{r :: s\}M \Rightarrow m} \tag{A.71}$$

Rules for judgement $\Rightarrow_d$

$$\frac{\begin{array}{c}(x' \Rightarrow n)\\ \vdots\end{array} \quad \begin{array}{c}\mathit{value}\ n \quad R' \Rightarrow_d m\end{array}}{[n/x]_d R \Rightarrow_d m}\ \mathsf{EC}(x, n, R, m) \tag{A.72}$$

$$\frac{M \Rightarrow m}{x = M \Rightarrow_d x \mapsto m} \quad \text{(A.73)}$$

$$\frac{R \Rightarrow_d r \quad S \Rightarrow_d s}{R \ \mathbf{and} S \Rightarrow_d r :: s} \quad \text{(A.74)}$$

$$\frac{R \Rightarrow_d r \quad \{r\}_d S \Rightarrow_d s}{R; S \Rightarrow_d r :: s} \quad \text{(A.75)}$$

$$\frac{R \Rightarrow_d r}{\{nil\}_d R \Rightarrow_d r} \quad \text{(A.76)}$$

$$\frac{[n/x]_d R \Rightarrow_d r}{\{x \mapsto n\}_d R \Rightarrow_d r} \quad \text{(A.77)}$$

$$\frac{\{r\}_d(\{s\}_d R) \Rightarrow_d r}{\{r :: s\}_d R \Rightarrow_d r} \quad \text{(A.78)}$$

Rules for judgement *closed*

$$\frac{closed\ M}{closed\ \{nil\}M} \quad \text{(A.79)}$$

$$\frac{closed\ [n/x]M}{closed\ \{x \mapsto n\}M} \quad \text{(A.80)}$$

$$\frac{closed\ \{r\}\{s\}M}{closed\ \{r :: s\}M} \quad \text{(A.81)}$$

$$\frac{closed\ m}{closed\ x \mapsto m} \quad \text{(A.82)}$$

$$\frac{R \gg m \quad closed\ \langle m \rangle M}{closed(\mathbf{let}\ R\ \mathbf{in}\ M)} \quad \text{(A.83)}$$

$$\frac{\begin{array}{c}(closed\ x)\\ \vdots \\ closed\ M\end{array}}{closed\ \langle x \rangle M} \quad \text{(A.84)}$$

$$\frac{closed\ \langle m \rangle(\langle n \rangle M)}{closed\ \langle m :: n \rangle M} \quad \text{(A.85)}$$

Rules for judgement $\gg$

$$\frac{}{\langle\rangle \gg nil} \quad \text{(A.86)}$$

$$\frac{R \gg m \quad S \gg n}{R \ \mathbf{and} S \gg m :: n} \quad \text{(A.87)}$$

$$\frac{\begin{array}{c}(closed\ x)\\ \vdots \\ closed\ M \quad R \gg n\end{array}}{x = M; R \gg x :: n} \quad \text{(A.88)}$$

A.1.3 NOS of $\mathcal{L}_{M_F}$

Rules for judgement $\Rightarrow$

$$\frac{}{\mathbf{struct}\ \ \mathbf{end} \Rightarrow nil} \quad \text{(A.89)}$$

$$\frac{M \Rightarrow m \quad [m/x]\mathbf{struct} B_{str} \Rightarrow l}{\mathbf{struct} x = M\ B_{str} \Rightarrow (x \mapsto m, l)} \quad \text{(A.90)}$$

$$\frac{M \Rightarrow m \quad N \Rightarrow t \quad proj\ m\ (t)\ n}{M : N \Rightarrow n} \quad \text{(A.91)}$$

$$\frac{u \Rightarrow m \quad (x \mapsto p)\ in\ m}{u.x \Rightarrow p} \quad \text{(A.92)}$$

$$\frac{u \Rightarrow l \quad \{l\}M \Rightarrow m}{\mathbf{open}\ u\ \mathbf{in} M \Rightarrow m} \quad \text{(A.93)}$$

$$\frac{\begin{array}{c}M \Rightarrow m \quad u \Rightarrow l\\ upd\ l\ x\ m\ l' \quad [u := l']N \Rightarrow n\end{array}}{[u.x := M]N \Rightarrow n} \quad \text{(A.94)}$$

Rules for judgement *value*

$$\frac{}{value(\mathbf{sig} B_{sig})} \quad \text{(A.95)}$$

Rules for judgement *closed*

$$\frac{}{closed\ \mathbf{sig} B_{sig}} \quad \text{(A.96)}$$

$$\frac{closed\ M \quad closed\ N}{closed\ M : N} \quad \text{(A.97)}$$

$$\frac{}{closed\ \mathbf{struct}\ \ \mathbf{end}} \quad \text{(A.98)}$$

$$\frac{\begin{array}{c}(closed\ x)\\ \vdots \\ closed\ M \quad closed(\mathbf{struct} B_{str})\end{array}}{closed(\mathbf{struct} x = M\ B_{str})} \quad \text{(A.99)}$$

$$\frac{closed\ u}{closed\ u.x} \quad \text{(A.100)}$$

$$\frac{closed\ u \quad closed\ M}{closed(\mathbf{open}\ u\ \mathbf{in} M)} \quad \text{(A.101)}$$

Rules for judgements *in*, *proj*, *upd*

$$\frac{}{m\ in\ (m :: l)} \tag{A.102}$$

$$\frac{m\ in\ l}{m\ in\ (p :: l)} \tag{A.103}$$

$$\frac{}{proj\ l\ (\mathbf{sig}\ \ \mathbf{end})\ nil} \tag{A.104}$$

$$\frac{(x \mapsto m)\ in\ l \quad proj\ l\ (\mathbf{sig}B_{sig})\ l'}{proj\ l\ (\mathbf{sig}x\ B_{sig})\ (x \mapsto m, l')} \tag{A.105}$$

$$\frac{}{upd\ (x \mapsto n, l)\ x\ m\ (x \mapsto m, l)} \tag{A.106}$$

$$\frac{upd\ l\ x\ m\ l'}{upd\ (y \mapsto n, l)\ x\ m\ (y \mapsto n, l')}\ x \neq y \tag{A.107}$$

A.1.4 NOS of $\mathcal{L}_{M_I}$

Rules for judgement *closed*

$$\frac{closed\ T}{closed\ T.f} \tag{A.108}$$

Rules for judgement $\gg$

$$\begin{array}{c}(closed\ T)\\ \vdots\\ \dfrac{R \gg I}{[p/T]_m R \gg I}\end{array} \tag{A.109}$$

Rules for judgement *free*

$$\frac{free\ R\ m \quad free\ M\ m}{free\ R.P(M)} \tag{A.110}$$

$$\frac{free\ R\ m}{free\ R.f\ m} \tag{A.111}$$

$$\frac{free\ p\ m \quad free\ N\ (R, m)}{free\ [p/R]_m N\ m} \tag{A.112}$$

$$\frac{\mathit{free}\ M\ m \quad \mathit{free}\ C\ (x,y) \quad \mathit{free}\ N\ (x) \quad \mathit{free}\ D\ (R,m)}{\mathit{free}\ (\textbf{module}\ R\ \textbf{is}\ x = M;\ \textbf{proc}\ P(y) = C;\ \textbf{func}\ f = N \textbf{in}\ D)\ m} \tag{A.113}$$

Rules for judgement $\Rightarrow$

$$\frac{\begin{array}{c}\begin{pmatrix} R' \Rightarrow_m (m, R') \\ (R', P) \Rightarrow_{mp} \boldsymbol{\lambda} x, y.C \\ (R', f) \Rightarrow_{mf} \boldsymbol{\lambda} x.N \end{pmatrix} \\ \vdots \\ M \Rightarrow m \quad \mathit{free}\ C\ (x,y) \quad \mathit{free}\ N\ (x) \qquad [D']N \Rightarrow n'\end{array}}{[\textbf{module}\ R\ \textbf{is}\ x = M_1;\ \textbf{proc}\ P(y) = C;\ \textbf{func}\ f = M_2 \textbf{in}\ D]N \Rightarrow n}\ \mathsf{EC}(R, D, m) \tag{A.114}$$

$$\frac{\begin{array}{c} R \Rightarrow_m (p, R') \qquad (R', P) \Rightarrow_{mp} \textbf{lambda}\ x, y.C \\ M \Rightarrow m \quad [p/x][m/y][C]x \Rightarrow p' \quad [p'/R]_m N \Rightarrow n \end{array}}{[R.P(M)]N \Rightarrow n} \tag{A.115}$$

$$\frac{\begin{array}{c}(R' \Rightarrow_m (p, T)) \\ \vdots \\ \mathit{value}\ p \quad R \Rightarrow_m (\_, T) \qquad N' \Rightarrow n'\end{array}}{[p/R]_m N \Rightarrow n}\ \mathsf{EC}(R, N, n) \tag{A.116}$$

$$\frac{R \Rightarrow_m (p, T) \quad (T, f) \Rightarrow_{mf} \textbf{lambda}\ x.M \quad [p/x]M \Rightarrow m}{R.f \Rightarrow m} \tag{A.117}$$

A.2 Denotational semantics

A.2.1 Denotational semantics of $\mathcal{L}_P$

Semantic domains

$$\begin{array}{rlrl} V &= (\mathcal{N} + \mathit{Truth} + U + P + F)^{\top}_{\bot} & P &= V \times V \\ \mathcal{N} &= \mathit{Nat}\ \text{(the domain of natural numbers)} & F &= V \to V \\ \mathit{Truth} &= \text{(the domain of truth values)} & \mathit{Env} &= ((\mathit{Id} \to V) \times (\mathit{ProcId} \to Q))^{\top} \\ U &= \mathit{Unit}\ \text{(the one-element domain)} & Q &= (\mathit{Id} \to V \to \mathit{Env} \to \mathit{Env})^{\top} \end{array}$$

Operators

$$\begin{array}{lll} \mathit{newenv} & = (\lambda x.\top, \lambda p.\top) & : \mathit{Env} \\ \mathit{update} & = \lambda x.\lambda n.\underline{\lambda}(\rho_v, \rho_p).([x \mapsto n]\rho_v, \rho_p) & : \mathit{Id} \to V \to \mathit{Env} \to \mathit{Env} \\ \mathit{access} & = \lambda x.\underline{\lambda}(\rho_v, \rho_p).\rho_v(x) & : \mathit{Id} \to \mathit{Env} \to V \\ \mathit{procupdate} & = \lambda p.\lambda q.\underline{\lambda}(\rho_v, \rho_p).(\rho_v, [p \mapsto q]\rho_p) & : \mathit{ProcId} \to Q \to \mathit{Env} \to \mathit{Env} \\ \mathit{procaccess} & = \lambda p.\underline{\lambda}(\rho_v, \rho_p).\rho_p(p) & : \mathit{ProcId} \to \mathit{Env} \to Q \\ \mathit{overlay} & = \underline{\lambda}\rho_1.\underline{\lambda}\rho_2.\lambda x.\textbf{if}\mathrm{is}\top(\rho_2(x)) \to \rho_1(x)\ [\!]\ \rho_2(x) & : \mathit{Env} \to \mathit{Env} \to \mathit{Env} \end{array}$$

Semantic functions

$$\mathcal{E} : Expr \to Env \to V \qquad \mathcal{D} : Decl \to Env \to Env$$
$$\mathcal{C} : Commands \to Env \to Env \qquad \mathcal{Q} : Proc \to Env \to Q$$

$$\mathcal{E}[\![x]\!] = \underline{\underline{\lambda}}\rho.access[\![x]\!]\rho \qquad \mathcal{E}[\![0]\!] = \underline{\underline{\lambda}}\rho.\text{in}\mathcal{N}(zero) \qquad \mathcal{E}[\![nil]\!] = \underline{\underline{\lambda}}\rho.\text{in}U()$$

$$\mathcal{E}[\![true]\!] = \underline{\underline{\lambda}}\rho.\text{in}Truth(true) \qquad \mathcal{E}[\![false]\!] = \underline{\underline{\lambda}}\rho.\text{in}Truth(false)$$

$\mathcal{E}[\![\textbf{let } x = M \textbf{ in } N]\!] = \underline{\underline{\lambda}}\rho.\textbf{let } v = \mathcal{E}[\![M]\!]\rho \textbf{ in } \mathcal{E}[\![N]\!](update\ [\![x]\!]\ v\ \rho)$

$\mathcal{E}[\![\textbf{letrec } f(x) = M \textbf{ in } N]\!] = \underline{\underline{\lambda}}\rho.\textbf{let } g = fix(\underline{\lambda}g.\underline{\lambda}v.\mathcal{E}[\![M]\!](update\ [\![f]\!]\ g\ \rho)) \textbf{ in}$
$\mathcal{E}[\![N]\!](update\ [\![f]\!]\ g\ \rho)$

$\mathcal{E}[\![\textbf{lambda}x.M]\!] = \underline{\underline{\lambda}}\rho.\text{in}F(\underline{\underline{\lambda}}v.\mathcal{E}[\![M]\!](update\ [\![x]\!]\ v\ \rho))$

$\mathcal{E}[\![M\ N]\!] = \underline{\underline{\lambda}}\rho.\textbf{cases } \mathcal{E}[\![M]\!]\rho \textbf{ of } \text{is}F(f) \to f(\mathcal{E}[\![N]\!]\rho)[\!]\ \top \textbf{end}$

$\mathcal{E}[\![M :: N]\!] = \underline{\underline{\lambda}}\rho.\textbf{let } v_1 = \mathcal{E}[\![M]\!]\rho \textbf{ in let } v_2 = \mathcal{E}[\![N]\!]\rho \textbf{ in } \text{in}P((v_1, v_2))$

$\mathcal{E}[\![[m/x]N]\!] = \underline{\underline{\lambda}}\rho.\textbf{let } v = \mathcal{E}[\![m]\!]\rho \textbf{ in } \mathcal{E}[\![N]\!](update\ [\![x]\!]\ v\ \rho)$

$\mathcal{E}[\![m \cdot n]\!] = \underline{\underline{\lambda}}\rho.\textbf{cases } \mathcal{E}[\![m]\!] \textbf{ of} \text{is}F(f) \to f(\mathcal{E}[\![n]\!]\rho)[\!]\ \top \textbf{end}$

$\mathcal{E}[\![[\textbf{on } \overline{x} = \overline{M} \textbf{ do } C]N]\!] = \underline{\underline{\lambda}}\rho.\textbf{if}(maxfree\ [\![C]\!](\overline{x})) \to \mathcal{C}[\![C]\!](\mathcal{D}[\![\overline{x} = \overline{M}]\!]\rho)[\!]\ \top$

where $maxfree : Commands \to Id^* \to Truth$; the meaning of "$maxfree\ [\![C]\!]\ s = true$" is simply "every free identifier of C is in s". *maxfree* is trivially defined on the syntactic structure of commands; we omit its definition.

$\mathcal{D}[\![\langle\rangle]\!] = \underline{\underline{\lambda}}\rho.newenv$

$\mathcal{D}[\![x = M; R]\!] = \underline{\underline{\lambda}}\rho.\textbf{let } v = \mathcal{E}[\![M]\!]\rho \textbf{ in}$
$\textbf{let } \tau = \mathcal{D}[\![R]\!](update\ [\![x]\!]\ v\ \rho) \textbf{ in}$
$overlay\ \tau\ (update\ [\![x]\!]\ v\ newenv)$

$\mathcal{D}[\![[n/x]_d R]\!] = \underline{\underline{\lambda}}\rho.\textbf{let } v = \mathcal{E}[\![n]\!]\rho \textbf{ in } \mathcal{D}[\![R]\!](update\ [\![x]\!]\ v\ \rho)$

$\mathcal{C}[\![x := M]\!] = \underline{\underline{\lambda}}\rho.\textbf{let } v = \mathcal{E}[\![M]\!]\rho \textbf{ in } update\ [\![x]\!]\ v\ \rho$

$\mathcal{C}[\![\textbf{while } M \textbf{ do } C]\!] = fix(F)$

where $F : (Env \to Env) \to (Env \to Env)$
$F = \underline{\lambda}f.\underline{\underline{\lambda}}\rho.\textbf{cases } \mathcal{E}[\![M]\!]\rho \textbf{ of}$
$\text{is}Truth(t) \to \textbf{if}t \to f(\mathcal{C}[\![C]\!]\rho)\ [\!]\ \rho$
$[\!]\ \top$
$\textbf{end}$

$\mathcal{C}[\![\textbf{begin new } x = M;\ C \textbf{ end}]\!] = \underline{\underline{\lambda}}\rho.\textbf{let } \rho' = update\ [\![x]\!]\ (\mathcal{E}[\![M]\!]\rho)\ \rho \textbf{ in}$
$\textbf{let } \rho'' = \mathcal{C}[\![C]\!]\rho' \textbf{ in}$
$update\ [\![x]\!]\ (access\ [\![x]\!]\ \rho)\ \rho''$

$\mathcal{C}[\![\textbf{procedure } P(x, y) = C \textbf{ in } D]\!] =$
$\underline{\underline{\lambda}}\rho.\textbf{let } \rho' = procupdate\ [\![P]\!]\ (\mathcal{Q}[\![\textbf{lambda}x, y.C]\!]\rho)\ \rho \textbf{ in}$
$\textbf{let } \rho'' = \mathcal{C}[\![D]\!]\rho' \textbf{ in}$
$procupdate\ [\![P]\!]\ (procaccess\ [\![P]\!]\ \rho)\ \rho''$

$$\mathcal{C}[\![P(x,M)]\!] = \underline{\underline{\lambda}}\rho.\textbf{let } v = \mathcal{E}[\![M]\!]\rho \textbf{ in } ((procaccess\ [\![P]\!]\ \rho)\ [\![x]\!]\ v\ \rho)$$

$$\begin{aligned}\mathcal{Q}[\![\textbf{lambda}x,y.C]\!] = \underline{\underline{\lambda}}\rho.&\textbf{if} maxfree\ [\![C]\!]\ ([\![x]\!],[\![y]\!]) emptysign)) \rightarrow\\ &\lambda i.\underline{\underline{\lambda}}v_y.\underline{\underline{\lambda}}\tau.\textbf{let } v_x = (access\ i\ \tau) \textbf{ in}\\ &\quad \textbf{let } \rho' = \mathcal{C}[\![C]\!](update\ [\![y]\!]\ v_y\ (update\ [\![x]\!]\ v_x\ \rho)) \textbf{ in}\\ &\quad\quad update\ i\ (access\ [\![x]\!]\ \rho')\ \tau\\ &[\!]\ \top\end{aligned}$$

$$\mathcal{E}[\![[Q/P]_{pe}M]\!] = \underline{\underline{\lambda}}\rho.\mathcal{E}[\![M]\!](procupdate\ [\![P]\!]\ \mathcal{Q}[\![Q]\!]\rho\ \rho)$$

A.2.2 Denotational semantics of $\mathcal{L}_D$

Semantic functions

$$\mathcal{O} : SyntEnvir \rightarrow Env \rightarrow Env\ \ \mathcal{E}[\![\textbf{let } R \textbf{ in } N]\!] = \underline{\underline{\lambda}}\rho.\mathcal{E}[\![N]\!](overlay\ (\mathcal{D}[\![R]\!]\rho)\ \rho)$$

$$\mathcal{E}[\![\{r\}M]\!] = \underline{\underline{\lambda}}\rho.\mathcal{E}[\![M]\!](\mathcal{O}[\![r]\!]\rho) = \mathcal{E}[\![M]\!] \circ \mathcal{O}[\![r]\!]$$

$$\mathcal{D}[\![R \textbf{ and} S]\!] = \underline{\underline{\lambda}}\rho.overlay\ (\mathcal{D}[\![S]\!]\rho)\ (\mathcal{D}[\![R]\!]\rho)$$

$$\mathcal{O}[\![x \mapsto n]\!] = \underline{\underline{\lambda}}\rho.update\ (\mathcal{E}[\![n]\!]\rho)\ \rho$$

$$\mathcal{O}[\![r :: s]\!] = \mathcal{O}[\![s]\!] \circ \mathcal{O}[\![r]\!]$$

A.2.3 Denotational semantics of $\mathcal{L}_{M_F}$

Semantic domains

$$\begin{aligned} V &= (\mathcal{N} + U + P + F + B + S)^{\top}_{\perp} & B &= Id \times V\\ U &= Unit & S &= Id^* = ES + CS\\ P &= V \times V & ES &= Unit\\ F &= V \rightarrow V & CS &= Id \times S\end{aligned}$$

Operators

$$\begin{aligned} emptystruct &= \text{in}U() &&: V\\ consstruct &= \lambda x.\underline{\underline{\lambda}}v.\underline{\underline{\lambda}}c.\text{in}P(\text{in}B([\![x]\!],v),c) &&: Id \rightarrow V \rightarrow V \rightarrow V\\ emptysign &= \text{in}ES() &&: V\\ consign &= \lambda i.\lambda t.\text{in}CS((i,t)) &&: Id \rightarrow S \rightarrow S\\ accessstruct &= \text{see below} &&: Id \rightarrow V \rightarrow V\\ applystruct &= \text{see below} &&: V \rightarrow Env \rightarrow Env\\ projection &= \text{see below} &&: V \rightarrow S \rightarrow V\\ longupdate &= \text{see below} &&: LongId \rightarrow V \rightarrow Env \rightarrow Env\end{aligned}$$

$$\begin{aligned} projection = \underline{\underline{\lambda}}s.\lambda t.&\textbf{cases } t \textbf{ of}\\ &\text{is}ES() \rightarrow \text{in}U()\\ &[\!]\ \text{is}CS(i,t') \rightarrow\\ &\quad \textbf{let } v = accessstruct\ i\ s \textbf{ in}\\ &\quad\quad \textbf{let } s' = projection\ s\ t' \textbf{ in } consstruct\ i\ v\ s'\\ &\textbf{end}\end{aligned}$$

Semantic functions

$$\mathcal{E}[\![\mathbf{structend}]\!] = \underline{\underline{\lambda}}\rho.emptystruct$$

$$\mathcal{E}[\![x \mapsto n]\!] = \underline{\underline{\lambda}}\rho.\mathbf{let}\ v = \mathcal{E}[\![n]\!]\rho\ \mathbf{in}\ \mathrm{in}B([\![x]\!], v)$$

$$\begin{aligned}\mathcal{E}[\![\mathbf{struct}x = M\ B_{str}]\!] = \underline{\underline{\lambda}}\rho.&\mathbf{let}\ v_1 = \mathcal{E}[\![M]\!]\rho\ \mathbf{in}\\ &\mathbf{let}\ v_2 = \mathcal{E}[\![\mathbf{struct}B_{str}]\!](update\ [\![x]\!]\ v_1\ \rho)\ \mathbf{in}\\ &\quad consstruct\ [\![x]\!]\ v_1\ v_2\end{aligned}$$

$$\mathcal{E}[\![\mathbf{sigend}]\!] = \underline{\underline{\lambda}}\rho.\mathrm{in}S(emptysign)$$

$$\begin{aligned}\mathcal{E}[\![\mathbf{sig}x\ B_{sig}]\!] = \underline{\underline{\lambda}}\rho.&\mathbf{cases}\ \mathcal{E}[\![\mathbf{sig}B_{sig}]\!]\rho\ \mathbf{of}\\ &\quad \mathrm{is}S(s) \to \mathrm{in}S(conssig\ [\![x]\!]\ s)\\ &\quad [\!]\ \top\\ &\mathbf{end}\end{aligned}$$

$$\begin{aligned}\mathcal{E}[\![M : N]\!] = \underline{\underline{\lambda}}\rho.&\mathbf{let}\ s = \mathcal{E}[\![M]\!]\rho\ \mathbf{in}\\ &\mathbf{cases}\ \mathcal{E}[\![N]\!]\rho\ \mathbf{of}\\ &\quad \mathrm{is}S(t) \to projection\ s\ t\\ &\quad [\!]\ \top\\ &\mathbf{end}\end{aligned}$$

$$\mathcal{E}[\![\mathbf{open}\ u\ \mathbf{in}M]\!] = \underline{\underline{\lambda}}\rho.\mathbf{let}\ s = \mathcal{E}[\![u]\!]\rho\ \mathbf{in}\ \mathcal{E}[\![M]\!](applystruct\ s\ \rho)$$

$$\mathcal{E}[\![u.x]\!] = \underline{\underline{\lambda}}\rho.accessstruct\ [\![x]\!]\ (\mathcal{E}[\![u]\!]\rho)$$

A.2.4 Denotational semantics of $\mathcal{L}_{M_I}$

Semantic domains

$$\begin{aligned} Env &= (IM \times PM \times MM)^\top & MM &= ModId \to M\\ IM &= Id \to V & M &= (V \times Q_M \times F_M)^\top\\ PM &= ProcId \to Q & Q_M &= V \to V \to V\\ Q &= (Id \to V \to Env \to Env)^\top & F_M &= V \to V = F\end{aligned}$$

Operators

$$\begin{aligned}
newenv &= (\lambda x.\top, \lambda p.\top, \lambda r.\top) && : Env\\
update &= \lambda x.\lambda n.\underline{\underline{\lambda}}(\rho_v, \rho_p, \rho_m).([x \mapsto n]\rho_v, \rho_p, \rho_m) && : Id \to V \to Env \to Env\\
access &= \lambda x.\underline{\underline{\lambda}}(\rho_v, \rho_p, \rho_m).\rho_v(x) && : Id \to Env \to V\\
procupdate &= \lambda p.\lambda q.\underline{\underline{\lambda}}(\rho_v, \rho_p, \rho_m).(\rho_v, [p \mapsto q]\rho_p, \rho_m) && : ProcId \to Q \to Env \to Env\\
procaccess &= \lambda p.\underline{\underline{\lambda}}(\rho_v, \rho_p, \rho_m).\rho_p(p) && : ProcId \to Env \to Q\\
modupdate &= \lambda r.\lambda q.\underline{\underline{\lambda}}(\rho_v, \rho_p, \rho_m).(\rho_v, \rho_p, [r \mapsto m]\rho_m) && : ProcId \to Q \to Env \to Env\\
modaccess &= \lambda r.\underline{\underline{\lambda}}(\rho_v, \rho_p, \rho_m).\rho_m(r) && : ProcId \to Env \to Q
\end{aligned}$$

Semantic functions

$$\begin{aligned}&\mathcal{C}[\![\mathbf{module}\ R\ \mathbf{is}\ x = M;\ \mathbf{proc}\ P(y) = C;\ \mathbf{func}\ f = N\mathbf{in}\ D]\!] =\\ &\quad \underline{\underline{\lambda}}\rho.\mathbf{if}maxfree\ [\![C]\!]\ (conssign\ [\![x]\!]\ (conssign\ [\![y]\!]emptysign)) \to\end{aligned}$$

$$
\begin{array}{l}
\quad\quad\quad\quad \textbf{if} \mathit{maxfree}\ [\![N]\!]\ (\mathit{conssign}\ [\![x]\!]\ \mathit{emptysign}) \rightarrow \\
\quad\quad\quad\quad\quad \textbf{let}\ m = \mathcal{E}[\![M]\!]\rho\ \textbf{in} \\
\quad\quad\quad\quad\quad\quad \textbf{let}\ q = \underline{\underline{\lambda}} v_x . \underline{\underline{\lambda}} v_y . \mathit{access}\ [\![x]\!]\ \mathcal{C}[\![C]\!](\mathit{update}\ [\![y]\!]\ v_y\ (\mathit{update}\ [\![x]\!]\ v_x\ \rho))\ \textbf{in} \\
\quad\quad\quad\quad\quad\quad\quad \textbf{let}\ g = \underline{\underline{\lambda}} v_x . \mathcal{E}[\![N]\!](\mathit{update}\ [\![x]\!]\ v_x\ \rho)\ \textbf{in} \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{let}\ \rho' = \mathit{modupdate}\ [\![R]\!]\ (m, q, g)\ \rho\ \textbf{in} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathit{modupdate}\ [\![R]\!]\ (\mathit{modaccess}\ [\![R]\!]\ \rho)\ \mathcal{C}[\![D]\!]\rho' \\
\quad\quad\quad\quad [\!]\ \top \\
\quad\quad\quad [\!]\ \top
\end{array}
$$

$$
\begin{aligned}
\mathcal{C}[\![R.P(M)]\!] = {} & \underline{\underline{\lambda}} \rho . \textbf{let}\ (n, q, g) = \mathit{modaccess}\ [\![R]\!]\ \rho\ \textbf{in} \\
& \quad\quad \textbf{let}\ n' = q\ n\ (\mathcal{E}[\![M]\!]\rho)\ \textbf{in} \\
& \quad\quad\quad \mathit{modupdate}\ [\![R]\!]\ (n', q, g)\ \rho
\end{aligned}
$$

$$
\mathcal{E}[\![R.f]\!] = \underline{\underline{\lambda}} \rho . \textbf{let}\ (n, q, g) = \mathit{modaccess}\ [\![R]\!]\ \rho\ \textbf{in}\ (g\ n)
$$

$$
\begin{aligned}
\mathcal{E}[\![[n/R]_m M]\!] = {} & \underline{\underline{\lambda}} \rho . \textbf{let}\ (\_, q, g) = \mathit{modaccess}\ [\![R]\!]\ \rho\ \textbf{in} \\
& \quad\quad \textbf{let}\ m' = \mathcal{E}[\![n]\!]\rho\ \textbf{in}\ \mathcal{E}[\![M]\!](\mathit{modupdate}\ [\![R]\!]\ (m', q, g)\ \rho)
\end{aligned}
$$

Appendix B

Proof of completeness of infinitary systems for Dynamic Logics

B.1 Proof of the Model Existence Lemma for First-Order Dynamic Logic

Let us recall the statement:

For A set of formulæ, if $A \nvdash \mathit{ff}$ then A is satisfiable.

The proof is an adaptation of an Henkin-like argument for infinitary logics, such as $L_{\omega_1\omega}$ [Har84]. In particular, we have to modify suitably the notions of *atoms* and *subformula closures*. Atoms are (possibly infinite) consistent sets of formulæ, while subformulæ are defined *á la* Fischer-Ladner, but with a different, infinitary treatment of the case of iteration: any finite iteration $[c^n]\,\varphi$, for $n \in \mathcal{N}$, is a subformula of $[c^*]\,\varphi$.

It is easier to adopt the "diamond" modal connective, in place of the "box". Therefore, in the rest of this chapter we will take $\langle c\rangle\,\varphi$ as primitive, while $[c]\varphi$ is derived. The rules for $\langle c\rangle\,\varphi$ are in Figure B.1. It is easy to prove that the rules for "diamond" and those for "box" are derivable/ammissible from each other, of course. Although the rules for assignment are truly Natural Deduction style, their side conditions are more easily described in linear (sequent-like) form:

$$:=\text{-I}\ \frac{\Gamma, y = t \vdash \varphi[y/x]}{\Gamma \vdash \langle x := t\rangle\,\varphi}\ y \notin \mathrm{FV}(\Gamma, \varphi, t)$$

$$:=\text{-E}\ \frac{\Gamma_1 \vdash \langle x := t\rangle\,\varphi \quad \Gamma_2, \varphi[y/x], y = t \vdash \psi}{\Gamma_1, \Gamma_2 \vdash \psi}\ y \notin \mathrm{FV}(\Gamma_2, \varphi, \psi, t)$$

Notice that also this version of **N***DL* is infinitary, but the infinitary rule is now the elimination of iteration.

Definition B.1 (Subformula) *Let $\varphi \in \Phi$ a formula of DL. The set of* subformulæ *of φ is the least $sub(\varphi) \subseteq \Phi$ such that the following closure properties hold:*

1. $\varphi \in sub(\varphi)$;

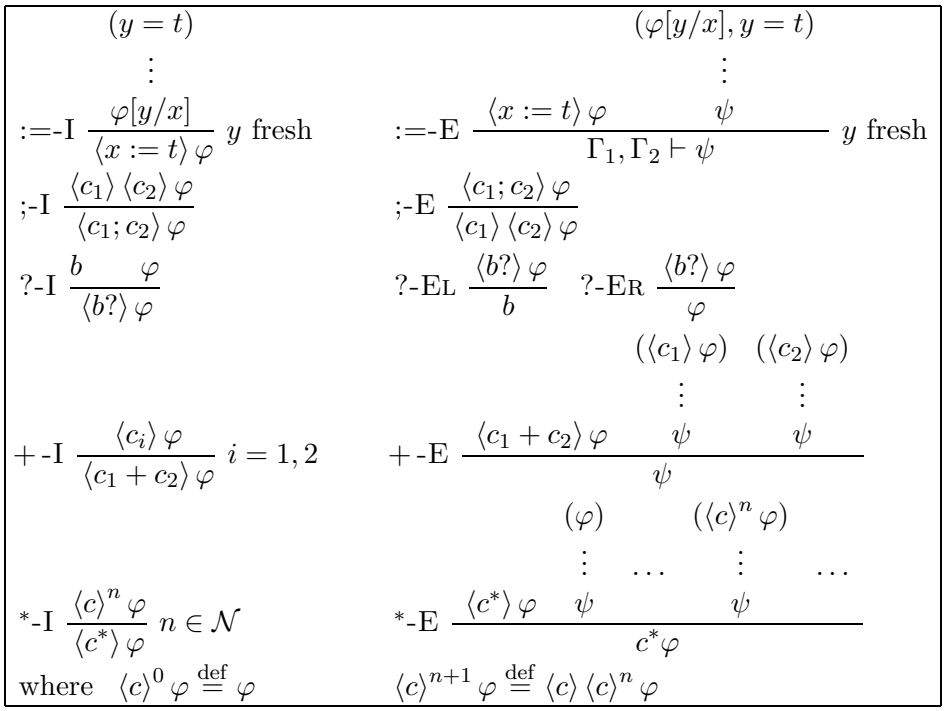

Figure B.1: The rules for the $\langle\cdot\rangle\cdot$ connective.

2. $\neg\psi \in sub(\varphi) \Rightarrow \psi \in sub(\varphi)$;
3. $\psi_1 \wedge \psi_2 \in sub(\varphi) \Rightarrow \psi_1, \psi_2 \in sub(\varphi)$;
4. $\psi_1 \supset \psi_2 \in sub(\varphi) \Rightarrow \psi_1, \psi_2 \in sub(\varphi)$;
5. $(\forall x\psi) \in sub(\varphi) \Rightarrow \psi \in sub(\varphi)$;
6. $\langle x := t\rangle\psi \in sub(\varphi) \Rightarrow \psi \in sub(\varphi)$;
7. $\langle c_1; c_2\rangle\psi \in sub(\varphi) \Rightarrow \langle c_1\rangle\psi, \langle c_2\rangle\psi \in sub(\varphi)$;
8. $\langle b?\rangle\psi \in sub(\varphi) \Rightarrow b, \psi \in sub(\varphi)$;
9. $\langle c^*\rangle\psi \in sub(\varphi) \Rightarrow$ *for all* $n \in \omega : \langle c\rangle^n\psi \in sub(\varphi)$.

The idea is that the subformulæ of φ are those formulæ whose truth values affect the value of φ, and are "simpler to derive" than φ.

Let G be an infinite set of new constant[1] symbols, ranged over by a, b. Let $DL(G)$ be the logic DL extended by adding G as terms. Symbols of G can be used everywhere we put a variable; for instance, $\langle a := 0\rangle\, a = 0$ is a legal formula. We adopt the convention

[1]Actually, the name "constant" is not very suited, since we allow for replacing variables by these constants also on the left-hand side of assignments. These new constants are used for denoting "free variables", but still having closed terms.

that symbols of G occurr only in free positions. We denote by $\vdash^G$ the derivation in this extended logic.

Notice that for Γ, φ in DL (that is, without symbols from G), we have that

$$\Gamma \vdash \varphi \quad \Longleftrightarrow \quad \Gamma \vdash^G \varphi$$

On one hand, trivially, every proof in DL is also a proof of $DL(G)$. On the other hand, let $\pi : (\Gamma \vdash^G \varphi)$; then, at most an enumerable set of symbols from G appear in π; we can replace these symbols by fresh variables, obtaining a proof in DL.

Therefore, the model existence lemma can be stated as follows:

$$\text{If } \Gamma \nvdash^G \mathit{ff}, \text{ then there is a model } \mathcal{M} \text{ for } \Gamma.$$

Definition B.2 (Atoms) *We define the set of* atoms *of $DL(G)$ as follows:*

$$At \stackrel{\text{def}}{=} \{A \subseteq DL(G) \mid A \nvdash^G \mathit{ff}, \text{ and the symbols from } G \text{ appearing in } A \text{ are finite}\}$$

It is evident that every set Γ of formulæ from DL is an atom.

We will prove that *every atom is satisfiable*, that is, given any $A_0 \in At$, there is a model $\mathcal{M}$ for A_0.

Definition B.3 (Closure) *Let $A_0 \in At$; we define the* closure *of A_0, $CL(A_0)$, as the least set of formulæ which satisfies the following closure properties:*

1. *for all $\varphi \in A_0 : sub(\varphi) \subseteq CL(A_0)$;*
2. *for all $a \in G, \varphi(t) \in CL(A_0) : \varphi(c) \in CL(A_0)$;*
3. *for all $a, b \in G : (a = b) \in CL(A_0)$.*

Let X be the set of closed formulæ of $CL(A_0)$ (possibly with symbols from G in place of free identifiers). X is enumerable (because $CL(A_0)$ is); let $X = \{\varphi_0, \varphi_1, \ldots\}$ an enumeration of X. Moreover, let T be the set of *basic terms* of $DL(G)$, that is, the terms of the form $f(a_1, \ldots, a_n)$. (In particular, $G \subseteq T$.) Also T is enumerable; let $(t_i)_i$ an enumeration.

Let us define a succession of atoms $(A_i)_{i \in \omega}$, such that A_0 is the given atom, and A_{i+1} is choosed such that the following properties hold:

1. $A_i \subseteq A_{i+1}$;
2. If $A_i, \varphi_i \nvdash \mathit{ff}$, then $\varphi_i \in A_{i+1}$. Moreover, if φ has a "choice" meaning, we need to include also a "witness" formula for one of the choices:
   - if $\varphi_i = \langle a := t \rangle \varphi$, then there is $a' \in G$ such that $A_i, a' = t, \varphi[a'/a] \nvdash^G \mathit{ff}$. Otherwise, suppose that for every $a' \in G$ we have $A_i, a' = t, \varphi[a'/a] \vdash^G \mathit{ff}$; then, we can choose a' fresh in A_i, t, φ, so we can replace a' by a fresh variable y, obtaining a proof of $A_i, y = t, \varphi[y/a] \nvdash^G \mathit{ff}$. But then, by applying :=-E, we obtain $A_i, \langle a := t \rangle \varphi \vdash \mathit{ff}$, a contradiction.
     Therefore, we put $\varphi[a'/a] \in A_{i+1}$.

- If $\varphi_i = \langle c_1 + c_2 \rangle \varphi$, then $A_i, \langle c_1 \rangle \varphi \nvdash^G \mathit{ff}$ or $A_i, \langle c_2 \rangle \varphi \nvdash^G \mathit{ff}$. Otherwise, by $+$-E, we obtain $A_i, \varphi_i \vdash^G \mathit{ff}$, a contradiction.
  Therefore, we put the appropriate $\langle c_j \rangle \varphi \in A_{i+1}$.
- If $\varphi_i = \langle c^* \rangle \varphi$, then there exists $n \in \omega$ such that $A_i, \langle c^n \rangle \varphi \nvdash^G \mathit{ff}$. Otherwise, suppose that for all $n \in \omega$, we have a proof of $A_i, \langle c^n \rangle \varphi \vdash^G \mathit{ff}$; then, by applying *-E we obtain $A_i, \varphi_i \vdash^G \mathit{ff}$, a contradiction.
  Therefore, we put $\langle c^n \rangle \varphi \in A_{i+1}$.

3. Furthermore, we require that there exists a formula $a = t_i$ in A_{i+1}, for some $a \in G$. Such formula can be already included in A_i, at some previous step; otherwise, we choose a fresh a and we add $a = t_i$ at this step.

Afer having defined the succession $(A_i)_{i \in \omega}$, we define

$$A_\omega \stackrel{\text{def}}{=} \bigcup_{i \in \omega} A_i$$

We define also a binary relation $\sim$ over G as follows:

$$a \sim b \iff (a = b) \in A_\omega$$

It is easy to see that $\sim$ is an equivalence:

- it is refexive: $a = a$ appears in X, and it is consistent with every consistent set; hence, it has been included in A_ω, at some step.
- it is symmetric: if $(a = b) \in A_\omega$, then for some i, $\varphi_i = (a = b)$, and φ is consistent with A_i; therefore, also $b = a$ is consistent with A_j, for $j \leq i$, so either $b = a$ is already in A_i, or it will be added at some step, later on. In both cases, $b = a$ appers in A_ω.
- it is transitive: let $a_1 = a_2, a_2 = a_3$ in A_ω; then, there is $m \in \omega$ such that $(a_1 = a_2) = \varphi_m$. Let $n \geq m$ such that $(a_1 = a_2), (a_2 = a_3) \in A_n$; therefore, by transitivity we have $A_n \vdash a_1 = a_3$. Then, we have also that $A_n, a_1 = a_3 \nvdash^G \mathit{ff}$, otherwise we would have $A_n \vdash \neg(a_1 = a_3)$, and by ff-I, $A_n \vdash \mathit{ff}$ — a contradiction.

Now we are ready for defining the *canonic model* for A_0. The model is a pair $\mathcal{M} = (D, \hat{\cdot})$ where

- Domain: $D \stackrel{\text{def}}{=} G/\sim = \{[a]_\sim \mid a \in G\}$;
- Interpretation of constants: $\hat{a} = [a]$;
- Interpretation of functional symbols:

$$\hat{f}(\hat{a_1}, \ldots, \hat{a_n}) = \hat{a} \iff (f(a_1, \ldots, a_n) = a) \in A_\omega$$

- Interpretation of relational symbols

$$\hat{R}(\hat{a_1}, \ldots, \hat{a_n}) \iff R(a_1, \ldots, a_n) \in A_\omega$$

This definition is well-given by the properties of congruence. For instance, if both the formula $(f(a_1, \ldots, a_n) = a)$ and the formula $(a_i = a_i')$ (for some $i = 1, \ldots, n$) are in A_ω, then also $(f(a_1, \ldots, a_i', \ldots, a_n) = a) \in A_\omega$.

Finally, we have to prove that $\mathcal{M}$ is actually a model for A_0. We prove that $\forall \varphi \in A_\omega : \mathcal{M} \models \varphi$; that is, $\mathcal{M}$ is a model for A_ω and, *a fortiori*, for A_0. We proceed by nested inductions on the structure of φ. The outside induction is on the order given by the definition of subformula.

Base Case: The base case is when φ is either atomic or the negation of an atomic formulæ. Let $\varphi = R(t_1, \ldots, t_n) \in A_\omega$, the other case being similar. Here, we need a nested induction on the structure of terms.

Base Case: The base case is therefore $R(a_1, \ldots, a_n)$, with $a_1, \ldots, a_n$ constants (from G or from DL). Then, by definition of $\mathcal{M}$, it is immediately $\mathcal{M} \models R(a_1, \ldots, a_n)$.

Inductive Step: For the sake of simplicity, but w.l.o.g., we take $n = 1$; that is, we have to prove that, take t term,

if for every t' simpler than t,

if $R(t') \in A_\omega$ then $\mathcal{M} \models R(t')$

then, if $R(t) \in A_\omega$ then $\mathcal{M} \models R(t)$.

Indeed, let $t = f(t_1, \ldots, t_k)$. Then, $\hat{t} = \hat{f}(\hat{t_1}, \ldots, \hat{t_k})$ is an object belonging to the domain of $\mathcal{M}$. Hence, there exists a such that $\hat{a} = \hat{t}$, and moreover $a_1, \ldots, a_k$ such that the formula $(f(a_1, \ldots, a_k) = c)$ appears in A_ω. By congruence rules, it is $R(f(a_1, \ldots, a_k)) \in A_\omega$. But the formula $f(a_1, \ldots, a_k)$ is simpler than t, therefore by inductive hypothesis we have $\mathcal{M} \models R(f(a_1, \ldots, a_k))$; by congruence, we obtain finally $\mathcal{M} \models R(f(t_1, \ldots, t_k))$.

Inductive Step: Let φ be a composed formulæ; we decompose it by following the definition of subformula. We see only significative cases.

- $\varphi = \varphi_1 \wedge \varphi_2$. Then, $\varphi_1, \varphi_2 \in sub(\varphi)$, and hence $\varphi_1, \varphi_2 \in X$; therefore, at some step both of them have been added to A_ω (otherwise, if $\varphi_1 \notin A_\omega$ then, at some step n, it is $A_n, \varphi_1 \vdash \mathit{ff}$, but then $A_n, \varphi_1 \wedge \varphi_2 \vdash \mathit{ff}$, a contradiction). By inductive hypothesis, we have $\mathcal{M} \models \varphi_1$ and $\mathcal{M} \models \varphi_2$, and hence the thesis.
- $\varphi = \langle c_1; c_2 \rangle \varphi_1$. Then, $\langle c_1 \rangle \langle c_2 \rangle \varphi_1 \in A_\omega$ (by an argument similar to the previous case). By inductive hypothesis, we have $\mathcal{M} \models \langle c_1 \rangle \langle c_2 \rangle \varphi_1$, and hence the thesis.
- $\varphi = \langle a := t \rangle \varphi_1$. Then, there exists a' such that $(a' = t), \varphi_1[a'/a] \in A_\omega$ (by an argument similar to the previous cases). By inductive hypothesis, we have $\mathcal{M} \models \varphi_1[a'/a]$, and moreover $\hat{a'} = \hat{t}$, and hence the thesis.
- $\varphi = \langle c^* \rangle \varphi_1$. Then, there exists n such that $\langle c \rangle^n \varphi_1 \in A_\omega$ (by an argument similar to the previous cases). By inductive hypothesis, we have $\mathcal{M} \models \langle c \rangle^n \varphi_1$, and hence the thesis.

Appendix C

Encoding of Validity FOL

Since Validity FOL does not satisfies the closure conditions, the naïf HOAS encoding is not correct:

```
Definition i := nat.

Inductive Set o :=
          Eq  : i -> i -> o
        | Imp : o -> o -> o
        | forall : (i -> o) -> o.

Token "=>". Infix 9 "=>" Imp.

Inductive V   : o -> Prop :=
          S   : (A,B,C:o)(V ((A => B => C) => ((A => B) => (A => C))))
        | K   : (A,B:o)(V (A => B => A))
        | MP  : (A,B:o)(V (A => B)) -> (V A) -> (V B)
        | Gen : (A:i -> o)(x:i)(V (A x)) -> (V (forall A)).
```

because it allows us for unsound derivations:

```
Lemma foo : (V  (Eq O O)) -> (V (forall [x:i](Eq x O))).
Intro; Apply Gen with x:=O; Assumption.
Qed.
```

Indeed, the correct implementation is the following, where `V` means “validity”:

```
Parameter var : Set.

Inductive i : Set :=
          Var  : var -> i
        | zero : i
        | plus : i -> i -> i.

Inductive o  : Set :=
          Eq  : i -> i -> o
        | Imp : o -> o -> o
```

```
        | forall : (var -> o) -> o.

Token "=>". Infix 9 "=>" Imp.

(* Two occur-check judgements *)
(* nii: not_in for terms *)
Fixpoint nii [x:var; t:i] : Prop :=
       <Prop>Case t of
               [y:var]~(x=y)
               True
               [t1,t2:i](nii x t1)/\(nii x t2)
             end.

(* nio: not_in for formulae *)
Fixpoint nio [x:var; A:o] : Prop :=
       <Prop>Case A of
               [t1,t2:i](nii x t1)/\(nii x t2)
               [A1,A2:o](nio x A1)/\(nio x A2)
               [B:var->o](y:var)~(x=y) -> (nio x (B y))
             end.

Inductive V : o -> Prop :=
          S : (A,B,C:o)(V ((A => B => C) => ((A => B) => (A => C))))
        | K : (A,B:o)(V (A => B => A))
        | MP : (A,B:o)(V (A => B)) -> (V A) -> (V B)
        | forall_I : (A:var -> o)(x:var)(nio x (forall A)) ->
                     (V (A x)) -> (V (forall A)).

Parameter x: var.

Lemma foo : (V (Eq (Var x) (Var x))) ->
            (V (forall [y:var](Eq (Var y) (Var y)))).
Intro. Apply forall_I with x:=x. Simpl. Intros.
(Split;Auto).
Assumption.
Qed.

Lemma bar : (V (Eq (Var x) zero)) ->
            (V (forall [y:var](Eq (Var y) zero))).
Intro. Apply forall_I with x:=x. Simpl. Intros.
(Split;Auto).
Assumption.
Qed.
```

Appendix D

A Methodology Roadmap for Encodings

In this chapter we summarize the methodology of representing faithfully a language $\mathcal{L}$ in a Logical Framework, described in more detail in Chapter 11. This chapter is shaped as a "flattened decision tree": the nesting of sections and subsections reflects the depth of a path in the decision tree.

D.1 Are there binding operators?

The main question is this: are there expression constructors whose arity are of level greater than 0?

D.1.1 No: plain inductive definitions

In this case, we can adopt a plain inductive definition, like those for free algebræ and propositional logic. Advantages: induction principles, easy adequacy theorems.

D.1.2 Yes! then, would you like some HOAS?

Let us consider the more interesting case of languages with binding operators, and hence with the equational theory induced by α-equivalence. As we have seen, the canonical approach is to represent the higher-order constructors by means of constants of higher-order type. However, straightforward HOAS may be not always feasible. In fact, it can clash with other features of the logic/language we are considering: closure of the language under substitution, schematicity of consequence relations, positivity constraints.

No, I do not need of HOAS!

A straightforward and easy-minded solution is to drop HOAS altogether, and to flatten every expression constructor to arities of level 0. For instance, the set Var of first-order variables ranging over i is represented by a specific type var, and a quantifier like $\forall$ is represented by a constant $forall : var \to o \to o$, whose arity is of level 0. (See e.g. [Ter95]).

Advantages: we can again adopt the plain inductive definitions, and hence we obtain the inductive principles for free. If we do not consider equational theory of our language (for instance, we consider $\forall x.\varphi$ different from $\forall y.\varphi[y/x]$), the adequacy theorems are easy to prove. In presence of subsorts, we do not need of "embeddings."

Disadvantages: we cannot delegate anything to the metalanguage, and mostly the α-equivalence and substitution properties. In fact, the representation is too fine-grained: two expressions of our language which are equivalent are represented by two terms which are not equivalent in the metalanguage. Therefore, we need to implement any equational theory (such as α-equivalence) and substitution mechanism (such as β-reductions) by hand.

Yes, thanks! I ♡ HOAS!

Binding operators are represented by means of constants of arity of level greater than 0, and the syntact expression $(x)e$ of arity $S \to S'$ is represented by means of the λ-abstraction of the metalanguage. For instance, the $\forall$ of FOL can be represented by a constant $forall : (var \to o) \to o$ (but this is not the only solution).

The immediate advantage is that we obtain the α-conversion for free, because we delegate it to the metalanguage. The disadvantage is that proofs of adequacy are more complex. Moreover, the introduction of an higher-order constructor on an inductive set can give rise to "exotic" terms, that is, terms which do not represent any phrase of the language. In some case, these have to be ruled out by means of extra judgements; see [DFH95].

Now, there is another choice: shall we define a specific type for identifiers, or shall we use the identifiers of the metalanguage as identifiers of the object language?

We define a specific type for variables. For instance, in first order logic we define $var, i, o : Set$ and $forall : (var \to o) \to o$.

Advantages: it can be applyed always (if the syntax and the consequence relations admit the right substitution schemata) we can use inductive definitions, obtaining induction principles, and the α-conversion is delegated to the metalanguage.

Disadvantage: substitution is not delegated to the metalanguage, so it has to be implemented on its own. There are more than one way for doing this, see next section.

We use metalanguage variables as object-level variables. For instance, in first order logic we define $i, o : Set$ and $forall : (i \to o) \to o$; in the λ-calculus, we define $lambda : (L \to L) \to L$.

Advantages: both α-conversion and substitution are delegated to the metalanguage. If the arity of constructors is nonnegative, we can also use inductive definitions, obtaining induction principles.

Disadvantages: if there are constructors of negative arity, we cannot adopt an inductive definition, and hence we miss inductive principles on the syntax. If the logical framework does not support inductive definitions, this is not a problem, since inductive principles cannot be defined anyway; in fact, this solution has been adopted in [AHMP92] for encoding several λ-calculi.

D.2 Need to implement substitution?

As we have seen in Chapter 11, we can face the problem of implementing a substitution mechanism "by hand". This happens when we have sorts with variables, and for some reason we cannot delegate the instantiation mechanism to the metalanguage. This is the case, for instance, of Dynamic Logic, NOS, Hoare Logic: the syntactic class is not closed under, or the consequence relation is not schematic with respect to, the substitution induced by the metalanguage. Or moreover, in the case of languages with negative constructors, such as the λ-calculus, and the μ-calculus.

There are two main ways for implementing a substitution mechanism: *inline* (immediate) or *delayed* (bookkeeping).

D.2.1 Inline (immediate) substitution

This solution is to implement directly the syntactic notion of substitution of the object language. This can be achieved by means of an *ad hoc* judgement, which represents the substitution mechanism. For instance, in the λ-calculus, the (nondeterministic) substitution $M[N/x]$ is represented by a judgement $subst : L \to (var \to L) \to L \to \mathsf{Prop}$: roughly, $(subst\ N\ (\lambda x{:}var.M)\ M')$ holds iff $M' = M[N/x]$. The rules for the derivation of *subst* correspond exactly to the cases of the object substitution mechanism. This approach has been adopted, for instance, in [DFH95].

Advantages: it is closer to the syntactic notion of substitution. If the set of identifiers has a decidable equality (e.g. is an inductive set), then the derivation of the *subst* judgement can be automated, for instace it can be defined as a function in the Coq environment.

Disadvantages: its use could be cumbersome. The user is loaded of the burden of realizing the substitution. Adequacy theorems are complex. In the case the set of identifier is inductive, there are exotic terms which have to be ruled out by some extra judgements (see [DFH95]).

D.2.2 Delayed substitution (bookkeeping)

The other way is to do not perform the substitution at all: we just keep track of the binding between the identifier and the expression by means of a bookkeeping assertion. These assertions are distributed in the derivation context, from which they are extracted on-the-fly, as soon as we have to evaluate an indentifier. This technique is inspired by the semantic idea of substitution, more than the syntactic one. See [BH90, Mic94, HM96], and Chapters 12, 14 and 15 for some applications.

Advantages: easier to use and understand; the implementation needs only an extra judgement (bookkeeping judgement) with no introduction rules, and the the evaluation rules for identifiers. It can be automated, at some extent, by means of the `Auto` tactic.

Disadvantages: it does not reflect the syntactic notion of substitution—in fact, it is *not* a substitution at all. Hence, adequacy is more difficult to state and prove. We need some extra judgement for implementing occur checking (although they can be mostly automatized).

A particular case: Leibnitz bookkeeping

If substitution is just textual replacement of terms for variables, like in FOL or λ-calculus, the bookkeeping technique can be fruithfully implemented by means of Leibnitz equality.

The idea is that the equality judgement can be used as the bookkeping judgement. Hence, we do not need to introduce any extra judgement or evaluation rules: the elimination principle of $=$ automatically provides powerful rules, not only for the evaluation of identifiers, but also for their replacing inside other terms. In the Coq environment, for instance, this can be done by means of the `Rewrite` tactic. This solution has been adopted in the encoding of μ-calculus (Chapter 15).

Disadvantages: it does not work for all logics, of course; for instance, in Dynamic Logic not all textual substitutions make sense. Secondly, the logical frameworks has to provide a structural congruence like Leibnitz equality; for instance, this is not the case for the LF. Moreover, we still need some extra judgements for occur checking.

Bibliography

[Abr87] Samson Abramsky. Domains in logical form. In *Second Annual IEEE Symposium on Logic in Computer Science*, pages 47–53. IEEE Computer Society Press, 1987.

[Acz94] Peter Aczel. Schematic consequence. In Dov Gabbay, editor, *What is a Logical System*. Oxford University Press, 1994.

[Add94] Bruno Adducci. Uno strumento integrato per lo sviluppo formale assistito di programmi: la formalizzazione di unity in un proof assistant. Master's thesis, Facoltà di Science Matematiche, Fisiche e Naturali, Universtià di Pisa, Italy, December 1994. In Italian.

[AGM92] Samson Abramsky, Dov Gabbay, and T. Maibaum, editors. *Handbook of Logic in Computer Science*. Oxford University Press, 1992.

[AHMP92] Arnon Avron, Furio Honsell, Ian A. Mason, and Robert Pollack. Using Typed Lambda Calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9:309–354, 1992.

[AHMP97] Arnon Avron, Furio Honsell, Marino Miculan, and Cristian Paravano. Encoding modal logics in Logical Frameworks. To appear, 1997.

[And93] Henrik Reif Andersen. *Verification of Temporal Properties of Concurrent Systems*. Daimi pb-445, Computer Science Department, Århus University, June 1993.

[AO91] Krzysztof R. Apt and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 1991.

[Apt81] Krzysztof R. Apt. Ten years of Hoare's logic: A survey — part I. *ACM Transactions on Programming Languages and Syms*, 3(4):431–483, October 1981.

[AS85] Harold Abelson and Gerard Jay Sussman. *Structure and Interpretation of Computer Programs*. The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, Massachusetts, 1985.

[Avr91] Arnon Avron. Simple consequence relations. *Information and Computation*, 92:105–139, January 1991.

[Bar92] Henk P. Barendregt. Lambda calculi with types. In Abramsky et al. [AGM92], pages 117–309.

[BB96] Janet Bertot and Yves Bertot. CtCoq: A system presentation. In Michael McRobbie and John Slaney, editors, *Automatic Deduction, CADE-13*, number 1104 in LNAI, pages 231–234. Springer-Verlag, July 1996.

[BC96] Stefano Berardi and Mario Coppo, editors. *Types for Proofs and Programs — Proceedings of the International Workshop TYPES'95*, number 1158 in Lecture Notes in Computer Science, Turin, March 1996. Springer-Verlag.

[Ber90] Stefano Berardi. *Type dependence and constructive mathematics*. PhD thesis, Mathematical Institute, Università di Torino, Italy, 1990.

[BG93] Mark Bezem and Jan Friso Groote, editors. *Proceedings of International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[BH90] Rod Burstall and Furio Honsell. Operational semantics in a natural deduction setting. In Huet and Plotkin [HP90], pages 185–214.

[BN94] Henk Barendregt and Tobias Nipkow, editors. *Proceedings of TYPES'93*, number 806 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

[CCF+95] Cristina Cornes, Judicaël Courant, Jean-Christophe Fillâtre, Gérard Huet, Pascal Manoury, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. *The Coq Proof Assistant Reference Manual - Version 5.10*. INRIA, Rocquencourt, July 1995. Available at `ftp://ftp.inria.fr/INRIA/coq/V5.10/doc/Reference-Manual.dvi.Z`.

[CDDK86] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *Proc. of the 1986 Conference on LISP and Functional Programming*. ACM Press, 1986.

[CDHL82] Mario Coppo, Mariangiola Dezani-Ciancaglini, Furio Honsell, and Giuseppe Longo. Extended type structures and filter lambda models. In G. Lolli, Giuseppe Longo, and A. Marcja, editors, *Proceedings of the Logic Colloquium '82*, pages 241–262, Amsterdam, 1982. North-Holland.

[Cen94] Pietro Cenciarelli. A modular development of denotational semantics. Presented at the *Types for Proofs and Programs* Annual Workshop, Båstad, Sweden, June 1994.

[Cer96] Iliano Cervesato. *A Linear Logical Framework*. PhD thesis, Dipartimento di Informatica, Università di Torino, Italy, 1996.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Control*, 76:95–120, 1988.

[Chi95] Jawahar Lal Chirimar. *Proof Theoretic Approach to Specification Languages.* PhD thesis, University of Pennsylvania, 1995.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Com94] Adriana B. Compagnoni. Subtyping in $f_{\omega\wedge}$ is decidable. Technical Report ECS-LFCS-94-281, LFCS, Department of Computer Science, University of Edinburgh, January 1994.

[Cou90] Patrick Cousot. Methods and Logics for Proving Programs. In van Leeuwen [vL90], pages 841–993.

[CP90] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and G. Mints, editors, *Proc. COLOG 88*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer-Verlag, 1990.

[dB80] Nicolas G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, New York, 1980.

[Des86] Joëlle Despeyroux. Proof of translation in natural semantics. In *Proceedings of the First Conference on Logic in Computer Science*, pages 193–205. The Association for Computing Machinery, 1986.

[DFH95] Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order syntax in Coq. In *Proc. of TLCA'95*, volume 905 of *Lecture Notes in Computer Science*, Edinburgh, April 1995. Springer-Verlag. Also appears as INRIA research report RR-2556, April 1995.

[DH94] Joëlle Despeyroux and André Hirschowitz. Higher-order syntax and induction in Coq. In *Proc. of LPAR'94*, Kiev, Ukraine, July 1994. Also appears as INRIA research report RR-2292, June 1994.

[Don77] James E. Donahue. Locations considered unnecessary. *Acta Informatica*, 8:221–242, July 1977.

[DPS96] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher order abstract syntax. Technical Report CMU-CS-96-172, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, August 1996.

[Dyb96] Peter Dybjer. Internal type theory. In Berardi and Coppo [BC96].

[EHR92] Lavinia Egidi, Furio Honsell, and Simona Ronchi-Della Rocca. Operational, denotational and logical descriptions: a case study. *Fundamenta Informaticæ*, 16(2):149–170, February 1992.

[Fel96] Mauro Felchero. Sistemi di transizione in teoria dei tipi coinduttivi. Master's thesis, Facoltà di Scienze Matematiche, Fisiche e Naturali, Università di Udine, Italy, July 1996. In Italian.

[Gar92] Philippa Gardner. *Representing Logics in Type Theory*. Cst-93-92, Department of Computer Science, University of Edinburgh, July 1992. Also published as ECS-LFCS-92-227.

[Gar93] Philippa Gardner. A new type theory for representing logics. In Andrei Voronkov, editor, *Proc. of LPAR '93*, volume 698 of *Lecture Notes in Computer Science*, pages 146–157, St. Petersburg, Russia, July 1993. Springer-Verlag.

[Gen69] Gerhard Gentzen. Investigations into logical deduction. In M. Szabo, editor, *The collected papers of Gerhard Gentzen*, pages 68–131. North Holland, 1969.

[Geu93] Jan Herman Geuvers. *Logics and Type Systems*. PhD thesis, Katholieke Universiteit, Nijmegen, The Nederlands, 1993.

[Gim95] E. Gimenez. Codifying guarded recursion definitions with recursive schemes. In Jan Smith, editor, *Proc. of TYPES'94*, Lecture Notes in Computer Science, Båstad, Sweden, 1995. Springer-Verlag.

[Gir87a] Jean-Yves Girard. Linear Logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gir87b] Jean-Yves Girard. *Proof Theory and Logical Complexity*. Bibliopolis, Napoli, 1987.

[GLT90] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.

[Gor88] Michael J. C. Gordon. HOL: A proof generating system for higher order logic. In Graham Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer Academic Press, 1988.

[Gor89] Michael J. C. Gordon. Mechanizing program logics in higher order logic. In P. A. Subrahmanyam and Graham Birtwistle, editors, *Current Trends in Hardware Verification and Automated Theorem Prover*, pages 387–439. Springer-Verlag, 1989.

[Gun92] Carl A. Gunter. *Semantics of programming languages: structures and techniques*. Foundations of Computing. The MIT Press, 1992.

[Han88] John J. Hannan. Proof-theoretical methods for analysis of functional programs. Technical Report MS–CIS–89–07, Dep. of Computer and Information Science, University of Pennsylvania, December 1988.

[Han93] John Hannan. Extended Natural Semantics. *Journal of Functional Programming*, 3(2):123–152, April 1993.

[Har79] David Harel. *First-Order Dynamic Logic*. Number 68 in Lecture Notes in Computer Science. Springer-Verlag, 1979.

[Har84] David Harel. Dynamic logic. In Dov Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume II, pages 497–604. Reidel, 1984.

[Har89] Robert Harper. Introduction to Standard ML. Technical Report ECS–LFCS–86–14, Department of Computer Science, University of Edinburgh, Edinburgh, Scotland, January 1989.

[Har90] Robert Harper. Systems of polymorphic type assignment in LF. Technical Report CMU-CS-90-144, Scool of Computer Science, Carnegie Mellon University, Pittsburgh, January 1990.

[HC84] George Edward Hughes and M. J. Cresswell. *A companion to Modal Logic*. Methuen, London, 1984.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[HM85] M. Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of ACM*, 32:137–162, 1985.

[HM93] Furio Honsell and Marino Miculan. Encoding program logics in type theories. In Joëlle Despeyroux, editor, *Deliverables of the TYPES Workshop* Proving Properties of Programming Languages, Sophia-Antipolis, September 1993.

[HM96] Furio Honsell and Marino Miculan. A natural deduction approach to dynamic logics. In Berardi and Coppo [BC96], pages 165–182. A preliminar version has been communicated to the *TYPES'94 Annual Workshop*, Båstad, July 1994.

[HMST93] Furio Honsell, Ian A. Mason, Scott Smith, and Carolyn Talcott. A Variable Typed Logic of Effects. *Information and Computation*, 1993.

[Hoa69] Christian Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[HP90] Gérard Huet and Gordon Plotkin, editors. *Logical Frameworks*. Cambridge University Press, June 1990.

[HP92] Robert Harper and Frank Pfenning. Compiler verification in LF. In *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992. IEEE Computer Society Press.

[Hue92] Gérard Huet. Constructive computation theory - part I. Lecture notes, October 1992.

[Hue94] Gérard Huet. Residual theory in λ-calculus: a formal development. *J. of Functional Programming*, 4,3:371–394, 1994.

[Kah87] Gilles Kahn. Natural Semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, number 247 in Lecture Notes in Computer Science, pages 22–39. Springer-Verlag, 1987.

[Koz83] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27, 1983.

[Koz95] Dexter Kozen, editor. *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, San Diego, California, 26–29 June 1995. The Institute of Electrical and Electronics Engineers, Inc., IEEE Computer Society Press.

[KP93a] Gilles Kahn and Gordon Plotkin. Concrete domains. *Theoretical Computer Science*, pages 187–277, December 1993.

[KP93b] Michael Kohlhase and Frank Pfenning. Unification in a λ-calculus with intersection types. In Dale Miller, editor, *Proc. of International Logic Programming Symposium*, pages 488–505, Vancouver, Canada, October 1993. The MIT Press.

[KT90] Dexter Kozen and Jerzy Tiuryn. Logics of Programs. In van Leeuwen [vL90], pages 789–840.

[Lam91] Leslie Lamport. The temporal logic of actions. Technical Report 79, Digital Equipment Corporation, System Research Center, 1991.

[LPT89] Zhaohui Luo, Robert Pollack, and Paul Taylor. *How to use LEGO (A Preliminary User's Manual)*. Department of Computer Science, University of Edinburgh, October 1989.

[LS88] Jacques Loeckx and Kurt Sieber. *The Foundations of Program Verification*. John Wiley and sons, New York, second edition, 1988.

[Luo96] Zhaohui Luo. Coercive subtyping in type theory. In *Proc. Annual Conf. of the European Association for Computer Science Logic*, Utrecht, 1996.

[Mar85] Per Martin-Löf. On the meaning of the logical constants and the justifications of the logic laws. Technical Report 2, Scuola di Specializzazione in Logica Matematica, Dipartimento di Matematica, Università di Siena, 1985.

[MG95] Marino Miculan and Fabio Gadducci. Modal μ-types for processes. In Kozen [Koz95], pages 221–231.

[MH82] Albert R. Meyer and Joseph Y. Halpern. Axiomatic definition of programming languages: A theoretical assessment. *Journal of the ACM*, 29(2):555–576, April 1982.

[Mic94] Marino Miculan. The expressive power of structural operational semantics with explicit assumptions. In Barendregt and Nipkow [BN94], pages 292–320.

[Mil94] Dale Miller. A multiple-conclusion meta-logic. In Samson Abramsky, editor, *Proceedings of the 9th LICS*, pages 272–281, Paris, July 1994. The Institute of Electrical and Electronics Engineers, Inc.

[MN94] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In Barendregt and Nipkow [BN94], pages 213–237.

[Mog93] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 1, 1993.

[Mor73] J. H. Morris, Jr. Types are not sets. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 120–124, Boston, October 1973. The Association for Computing Machinery.

[MP81] Zohar Manna and A. Pnueli. Verification of concurrent programs: the temporal framework. In Robert Boyer and J. Moore, editors, *The Correctness Problem in Computer Science*, pages 215–273. Academic Press, 1981.

[MP91] Spiro Michaylov and Frank Pfenning. Natural Semantics and some of its Meta-Theory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, number 596 in LNAI, pages 299–344, Stockolm, Sweden, January 1991. Springer-Verlag.

[MP93] James McKinna and Robert Pollack. Pure type systems formalized. In Bezem and Groote [BG93], pages 289–305.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[NP92] Tobias Nipkow and Lawrence C. Paulson. Isabelle-91. In D. Kapur, editor, *Proc. of the 11th CADE*, number 607 in Lecture Notes in Computer Science, pages 673–676, Saratoga Springs, NY, 1992. Springer-Verlag. System abstract.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*, volume 7 of *International Series of Monograph on Computer Science*. Oxford University Press, 1990.

[NPS92] Bengt Nordström, Kent Petersson, and Jan M. Smith. Martin-löf's type theory. In Abramsky et al. [AGM92].

[Pau89] Christine Paulin-Mohring. Extracting f_ω's programs from proofs in the Calculus of Constructions. In *Proc. 16th PoPL*, Austin, January 1989. The Association for Computing Machinery.

[Pau93] Christine Paulin-Mohring. Inductive definitions in the system Coq; rules and properties. In Bezem and Groote [BG93], pages 328–345.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proc. of ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988. The Association for Computing Machinery.

[Pfe89] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322. The Institute of Electrical and Electronics Engineers, Inc., June 1989. Also available as ERGO Report 89–067, School of Computer Science, Carnegie Mellon Univ., Pittsburgh.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. DAIMI FN-19, Computer Science Department, Århus University, Århus, Denmark, September 1981.

[Plo85] Gordon D. Plotkin. Notes about semantics. Unpublished notes given at CSLI, Stanford, August 1985.

[Pra65] Dag Prawitz. *Natural Deduction*. Almqvist & Wiksell, Stockholm, 1965.

[PW90] David Pym and Lincoln Wallen. Proof-search in the $\lambda\pi$-calculus. In Huet and Plotkin [HP90], pages 309–340.

[PW93] Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[PW95] Frank Pfenning and Hao-Chi Wong. On a modal λ-calculus for S4. In *Proc. MFPS'95*, 1995.

[Rey78] John C. Reynolds. Syntactic control of interference. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 39–46, Tucson, October 1978. The Association for Computing Machinery.

[Saï96] Amokrane Saïbi. Axiomatization of a λ-calculus with explicit substitutions in coq. In Berardi and Coppo [BC96].

[Sca97] Ivan Scagnetto. π-calculus in CIC. Forthcoming master thesis, Università di Udine, 1997.

[Sch86] David A. Schmidt. *Denotational Semantics*. Allyn & Bacon, 1986.

[Sch94] David A. Schmidt. *The Structure of Typed Programming Languages*. Foundations of Computing. MIT, Cambridge, MA, 1994.

[Sco82] Dana Scott. Domains for denotational semantics. In *Proceedings of the ICALP 1982*, number 140 in Lecture Notes in Computer Science, pages 577–613. Springer-Verlag, 1982.

[Sti85] Colin Stirling. Logics for While Programs: Algorithmic/Dynamic Logics. Unpublished notes, 1985.

[Sti92] Colin Stirling. Modal and Temporal Logics. In Abramsky et al. [AGM92], pages 477–563.

[SW89] Colin Stirling and David Walker. Local model checking for the modal μ-calculus. In *Proceedings of the TAPSOFT Conference*, 1989.

[Tay88] Paul Taylor. Using Constructions as a metalanguage. Technical Report ECS-LFCS-88-70, Department of Computer Science, University of Edinburgh, December 1988.

[Ten81] Robert D. Tennent. *Principles of Programming Languages*. Prentice-Hall, London, 1981.

[Ter95] Delphine Terrasse. Encoding Natural Semantics in Coq. In *Proc. AMAST'95*, number 936 in Lecture Notes in Computer Science, pages 230–244, 1995.

[vB83] Johan van Benthem. *Modal logic and classical logic*, volume 3 of *Monographs in philosophical logic and formal linguistics*. Bibliopolis, Napoli, 1983.

[vL90] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science*. North Holland, 1990.

[Wal95a] Igor Walukiewicz. Completeness of Kozen's axiomatisation. In Kozen [Koz95], pages 14–24.

[Wal95b] Igor Walukiewicz. Notes on the propositional μ-calculus: Completeness and related results. Notes Series NS-95-1, BRICS, Department of Computer Science, University of Aarhus, Denmark, February 1995.

[Wer94] Benjamin Werner. *Une théorie des constructions inductives*. PhD thesis, Université Paris 7, 1994.

[WF91] Andrew K. Wright and Matthias Fellaisen. A syntactic approach to type soundness. Technical Report TR91–160, rev.2, Department of Computer Science, Rice University, Houston, Texas, 1991.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, 1993.

About this document

The source file of this thesis amounts to more than 585 kbytes of LaTeX code (Coq code is excluded from this count). I adopted the class `book` at size 11pt, with `textwidth`=150mm, `textheight`=220mm, `oddsidemargin`=1cm, `evensidemargin`=2mm, `headheight`=14pt, `footskip`=25pt. The running headers have been obtained by using Piet van Oostrum's fancyhdr package, version 1.99b. The title page has been designed on my own.

Almost all the time I worked on a Macintosh SE/30 (8 Mbytes RAM, 49 Mbytes HD). I begun editing this thesis by using GNU Emacs 19.29.1 with AUCTeX 9.4 on `ten.dimi.uniud.it`, a SPARCserver-20 (later upgraded to a four-processor SPARCserver-1000) running SunOS 5.5; the telnet connection has been provided by NCSA Telnet 2.6, through a LocalTalk LAN. However, the unsteadyness of LAN performances (swinging from 16 to 2 kbytes/sec, and less) yielded me to move the editing activity locally on the Macintosh, where (like in this moment) I used Marc Parmet's port of GNU Emacs 18.59.

Compilations took always place on `ten`, where the source file was transferred by means of the `ftp` server built-in NCSA Telnet 2.6. The `latex` compiler (based on TeX 3.1415, C version 6.1, and LaTeX 2_ε of December 1, 1995, patch level 2), took 52 seconds to compile the whole document.

The resulting 885 kbytes DVI file has been printed (like every preliminary version, since an A4 page poorly fits a 9" Mac screen) on an Apple LaserWriter 16/600. The Postscript file, produced by dvips 5.528a, takes 4.69 Mbytes. Could such a heavy duty have been the reason for those frequent paper jams? I do not know—anyway, these faults occurred *so* often that the printer has been replaced by another one, of the same model (and, sadly enough, with the same performances).

Like a multitude of scientists all around the world (and beyond), I owe a great debt to Donald Knuth, Leslie Lamport and the whole LaTeX3 Project team, for their invaluable effort in developing the TeX and LaTeX systems. These immensely successful and elegant applications of computers acquainted me with one of the brightest yet neglected expressions of Western Culture: the sublime Art of Typography, whose roots founder in dusty *scriptoria* dwelt by thousands of medioeval amanuenses and illuminators—will their precious work be never forgotten.