Università degli Studi di Udine
Dipartimento di Matematica e Informatica
Dottorato di Ricerca in Informatica

Institut National Polytechnique de Lorraine
École Nationale Supérieure des Mines, Nancy
Laboratoire Lorraine de Recherche en Informatique et ses Applications
Doctorat de Recherche en Informatique

Ph.D. Thesis

Certified Reasoning on Real Numbers and Objects in Co-inductive Type Theory

Candidate:
Alberto Ciaffaglione

Supervisors:
Furio Honsell
Pietro Di Gianantonio
Claude Kirchner
Luigi Liquori

Author's e-mail: ciaffagl@dimi.uniud.it, Alberto.Ciaffaglione@loria.fr

Author's address:

Dipartimento di Matematica e Informatica
Università degli Studi di Udine
via delle Scienze, 206
33100 Udine (Italia)

Abstract

In this thesis we adopt Formal Methods based on Type Theory for reasoning on the semantics of computer programs. The ultimate goal is their certification, that is, to prove that a fragment of software meets its formal specification.

Application areas of our research are the representation of and computation on the Real Numbers and the Object-oriented Languages based on Objects. In our investigation, Coinductive specification and proof principles play an essential role.

In the first part of the thesis we deal with Constructive Real Numbers. We construct the reals using streams, i.e. infinite sequences, of signed digits. We work with constructive logic, that is, we develop "algorithmic" mathematics. We implement the Real Numbers in Coq using streams, which are managed using coinductive judgments and corecursive algorithms. Then we introduce a constructive axiomatization and we use it for proving the adequacy of our construction.

In the second part of the thesis we approach Object-based Calculi with side-effects, in order to address important properties, such as Type Soundness, formally. We state our investigation focusing on Abadi and Cardelli's **imp**ς-calculus, which features objects, cloning, dynamic lookup, imperative method update, object types and subsumption. We reformulate **imp**ς using modern encoding techniques, as Higher-Order Abstract Syntax and Coinductive proof systems in Natural Deduction style. Then we formalize **imp**ς in Coq and we prove the Type Soundness.

To my parents,
Elisa, and Alessia

Acknowledgments

I wish to thank many people, who helped me in the course of my PhD programme in Udine and Nancy.

First of all, I am grateful to my supervisors, Furio Honsell and Pietro Di Gianantonio in Udine, Claude Kirchner and Luigi Liquori in Nancy, for their guidance, inspiring comments and useful suggestions.

I benefited a lot of the collaboration with Marino Miculan, Ivan Scagnetto and Marina Lenisa at the Department of Mathematics and Computer Science in Udine. A special thank to the people of the Protheo group at LORIA, who allowed me to make more profitable my stay in Nancy. And also thanks to Joëlle Despeyroux, Bernard Serpette and Yves Bertot, for interesting discussions during the month spent in Sophia Antipolis.

Finally, my greatest gratitude is devoted to my family and my girlfriend, for fully supporting me along the years of this adventure.

April 30, 2003

Alberto Ciaffaglione

Contents

Introduction and Overview iii

Résumé vii

I Preliminary reflections 1

1 Formal Methods based on Type Theory 3
1.1 Computational metamodels . . . 3
1.2 Logical frameworks based on typed λ-calculus . . . 4
1.3 The Calculus of (Co)Inductive Constructions . . . 9
1.4 The proof assistant Coq . . . 12
1.5 Circularity and Coinduction . . . 19

2 Focusing on Real Numbers and Objects 25
2.1 Motivations for Real Numbers . . . 25
2.2 Constructions of the Real Numbers . . . 27
2.3 Exact computation . . . 28
2.4 Constructive Mathematics, Computer Science . . . 29
2.5 Motivations for Objects . . . 30
2.6 Formal Methods and Objects . . . 31

II Real Numbers 33

3 A co-inductive model of the Real Numbers in Coq 35
3.1 Real numbers in logical frameworks . . . 35
3.2 Construction via streams . . . 36
3.3 Adequacy of the encoding . . . 41
3.4 Formalization in Coq . . . 43
3.5 Redundancy and equivalence . . . 48
3.6 Certification of exact algorithms . . . 51
3.7 Related work . . . 57

4 Axiomatizations of Constructive Real Numbers 59
4.1 A minimal constructive axiomatization . . . 59
4.2 Consistency of the axioms . . . 63

4.3 Axioms at work . . . 74
4.4 Equivalent axiomatizations . . . 76
4.5 Completeness and categoricity . . . 81
4.6 Conclusion . . . 82

III Objects **85**

5 Abadi and Cardelli's impς-calculus **87**
5.1 Overview of the calculus . . . 87
5.2 Examples . . . 91
5.3 Type Soundness . . . 94

6 Reformulation of impς in Natural Deduction **99**
6.1 Proof-theoretical choices . . . 99
6.2 Dynamic and static semantics . . . 102
6.3 Adequacy . . . 109
6.4 Preliminary metatheory . . . 112
6.5 Result typing by coinduction . . . 113
6.6 Result typing by induction . . . 119

7 A Type Sound formalization of impς in Coq **131**
7.1 Encoding methodology . . . 131
7.2 Formalization in Coq . . . 132
7.3 Metatheory in Coq . . . 147
7.4 Type Soundness . . . 151
7.5 Related work . . . 152
7.6 Conclusion . . . 152

Bibliography **155**

Introduction and Overview

Nowadays there is a strong need for dependable Information Technology, because computer science is pervasive throughout all areas of human activity, and computerized systems are ubiquitous. Thus software failures should be avoided, because they may cause serious consequences both to building firms and users, even loss of life in the case of safety-critical applications, as for example car engine management systems, heart pacemakers, radiation therapy machines, nuclear reactor controllers, fly-by-wire aircrafts, and so on.

Even if absolute validation is utopian, we can achieve a greater reliability on programs and systems, thus increasing our confidence, through a rigorous analysis based on formal proofs, and using logics validated by mathematical models.

This goal is approached in the present thesis by the use of Constructive Type Theory, viz Typed Lambda Calculus. Such a (special-purpose) formal system is an excellent tool for representing formal reasoning, and is the basis of the computational metamodel of Type Theory-based Logical Frameworks (LFs), namely general specification environments where logic-independent reasoning is possible, and which permit to factorize out the differences across object systems. Hence LFs are very useful for encoding generic calculi and logics, for developing prototypes, and then reasoning on specifications and models by formal proofs.

LFs are based on the Curry-Howard-de Bruijn (or propositions-as-types) [How80, dB80] analogy, i.e. a close correspondence between types-propositions and λterms-proofs. Types can be actually interpreted as propositions, and terms inhabiting types are exactly the proofs of the associated propositions. Hence formal proofs have computational content, because they embed algorithms, and typed functional languages can be seen as languages for proofs. Proceeding further, a proposition is true if the corresponding type is inhabited, thus proof checking reduces to type checking. In the case that type checking is decidable, LFs become a suitable basis for implementing interactive proof assistants (Nuprl, Lego, Coq, Alf, etc.).

Interactive proof assistants are systems where "the human provides the intelligence and the automated system does part of the craftsmanship". They actually allow to combine proof building —which is creative and difficult to automate— with proof checking— which is, on the contrary, routine, and can be an highly error-prone activity for the human mind. Among interactive proof assistants, the semi-automated ones, as for example Coq [INR03], seem a good compromise. The system Coq, which is based on the Calculus of Constructions [CH88], interpreter of the proposition-as-types principle for the higher-order intuitionistic logic, plays a prominent role in the present thesis.

Aims of the thesis. In this thesis we adopt Formal Methods based on Type Theory for reasoning on the semantics of computer programs. The ultimate goal is their certification, that is, to prove that a fragment of software meets its formal specification.

Application areas of our research are the representation of and computation on the Real Numbers and the Object-oriented Programming Languages based on Objects. In our investigation, coinductive specification and proof principles play an essential role.

Induction and coinduction are different tools for reasoning on circularity, or self reference. The human process of rational and formal reasoning needs the use of circularity, not only when considering computer science, mathematics or logic, but also in many other disciplines. Self reference is pervasive in theoretical computer science, because it is extremely natural and powerful for modeling and studying the behavior and the semantics of computing systems. Among circular phenomena, it is natural to distinguish between inductive and coinductive ones: induction supplies principles for defining and reasoning on circular, finite objects; coinduction is quite well-suited with respect to circular, non well-founded objects.

Some researchers capture this distinction from a more abstract point of view, using a categorical terminology: in this perspective, it is possible to state a duality between algebras —involved in the description of abstract data types— and coalgebras —which arise in the analysis of state-based dynamical structures occurring in computer science. Many applicative situations fall in the coalgebraic, or coinductive, area: typically all those phenomena where an infinite amount of information is involved (e.g. never terminating processes, streams, exact real numbers, loops of pointers in the memory, etc).

Synopsis. The present document is organized in three distinct parts. The first one supplies the background useful for developing the other two. More precisely, chapter 1 focus on the tools we work with, namely Type Theory-based Logical Frameworks, the proof assistant Coq and coinductive principles. The thesis really starts in chapter 2, where we collect the motivations for the work we carry out in the following chapters.

The second part of the thesis deals with constructive real numbers. We construct the reals in chapter 3 using streams, i.e. infinite sequences, of signed digits. We work with constructive logic, that is, we develop "algorithmic" mathematics, in the sense of Bishop [Bis67] and Martin-Löf [ML82]. We implement real numbers in Coq as pairs naturals-streams, which are managed using coinductive judgments and corecursive algorithms. In chapter 4 we introduce a constructive axiomatization of the reals and we use it for proving the adequacy of our construction. Therefore, we have built reliable real numbers and we have certified the (arithmetic) algorithms working on the implementation. Moreover, the functions implementing the algorithms can be extracted from Coq, thus providing certified software.

The third part of the thesis is independent from the second one. We approach formally object-based calculi with side-effects with the aim of addressing important (meta)theoretical properties, as e.g. Type Soundness. This is a first effort towards the synthesis of certified software for object-based languages. We state our investigation focusing on Abadi-Cardelli's $\mathbf{imp}\varsigma$-calculus [AC96], a very representative calculus featuring objects, cloning, dynamic lookup, imperative method update, object types and subsumption. In chapter 5 we survey $\mathbf{imp}\varsigma$, which is after reformulated in chapter 6 using modern encoding techniques, as Higher-Order Abstract Syntax (HOAS) and coinductive proof systems in Natural Deduction style. These proof systems are used here for the first time in combination with HOAS for dealing with an object-based calculus and proving a Type Soundness theorem. Finally, in chapter 7, we formalize $\mathbf{imp}\varsigma$ in Coq and we prove formally the Type

Soundness, taking full advantage of the techniques provided by $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$, the coinductive type theory underlying Coq.

Résumé

Il y a de nos jours un besoin fort de Technologie sûre de l'Information, parce que l'informatique est dominante dans tous les secteurs d'activité humaine, et les systèmes automatisés sont omniprésents. Ainsi des échecs de logiciel devraient être évités, parce qu'ils peuvent causer des conséquences graves aux sociétés de bâtiment et aux utilisateurs, même perte de la vie dans le cas des applications sûreté-critiques, comme par exemple des systèmes de gestion de moteur de voiture, stimolateurs de coeur, machines de thérapie radiologique, contrôleurs de réacteur nucléaire, avions de voler-par-fil, et ainsi de suite.

Sur même si la validation absolue est utopique, nous pouvons obtenir une plus grande fiabilité des programmes et des systèmes, de ce fait augmentant notre confiance, par une analyse rigoureuse basée sur les preuves formelles, et employant des logiques validées par des modèles mathématiques.

Ce but est approché dans la thèse actuelle par l'utilisation de la Théorie Constructive de Types, c.-à-d. Lambda Calcul Typé. Un tel système formel est un excellent outil pour représenter le raisonnement formel, et est la base du metamodel informatique des Logical Frameworks basés sur la Théorie de Types (LFs), à savoir environnements généraux de spécifications où du raisonnement logique-indépendant est possible, et qui laissent factoriser hors des différences à travers des systèmes d'objet. Par conséquent LFs sont très utiles pour coder les calculs et les logiques génériques, pour developper des prototypes, et puis raisonner sur ces spécifications et modèles par des preuves formelles.

LFs sont basés sur l'analogie de Curry-Howard-De Bruijn (ou proposition-comme-types) [How80, dB80], c.-à-d. une correspondance étroite entre les types-propositions et les λterms-preuves. Les types peuvent être interprétés réellement comme propositions, et les habitants des types sont exactement les preuves des propositions associées. Par conséquent les preuves formelles ont du contenu informatique, parce qu'elles incluent des algorithmes, et les langages fonctionnelles typés peuvent être vues comme langages pour des preuves. Procédant plus loin, une proposition est vraie si le type correspondant est habité, ainsi la vérification de preuve réduit à la vérification de type. Dans le cas qui la vérification de type l'on peut décider, LFs deviennent une base appropriée pour les aides interactifs de preuve (Nuprl, Lego, Coq, Alf, et ainsi de suite).

Les aides interactifs de preuve sont des systèmes où le "l'humain fournit l'intelligence et le système automatisé partie de l'art". Ils laissent réellement combiner le bâtiment de preuve —qui est créateur et difficile à automatiser— avec la vérification de preuve —qui est, au contraire, courant, et peut être une activité fortement erreur-encline pour l'esprit humain. Parmi les aides interactifs de preuve, les semi-automatisés, en tant que par exemple Coq [INR03], semblent un bon compromis. Le système Coq, qui est basé sur le Calcul des Constructions [CH88], interprète du principe de proposition-comme-types pour la logique intuitionniste d'haut-ordre, jeux un rôle en avant dans la thèse actuelle.

Objectifs de la thèse. Dans cette thèse nous adoptons Méthodes Formelles basées sur la Théorie de Types pour raisonner sur la sémantique des programmes machine. Le but final est leur certification, c.-à-d., montrer qu'un fragment de logiciel répond à ses spécifications formelles.

Les domaines d'application de notre recherche sont la représentation de et le calcul sur les Nombres Réels et les Langages de Programmation Orientés aux Objets basés sur les objets. Dans notre recherche, les principes coinductifs de spécification et de preuve jouent un rôle essentiel.

L'induction et la coinduction sont différents outils pour raisonner sur le circularité, ou référence si même. Le processus humain de raisonnement raisonnable et formel a besoin de l'utilisation de la circularité, non seulement en considérant l'informatique, les mathématiques ou la logique, mais également dans beaucoup d'autres disciplines. La référence si même est dominante dans l'informatique théorique, parce qu'elle est extrêmement normale et puissante pour modeler et étudier le comportement et la sémantique des systèmes de calcul. Parmi des phénomènes circulaires, il est normal de distinguer entre les inductifs et les coinductifs : l'induction fournit des principes pour définir et raisonner sur des objets circulaires et finis ; la coinduction est bien adaptée en ce qui concerne les objets circulaires et non bien fondés.

Quelques chercheurs capturent cette distinction d'un point de vue plus abstrait, en utilisant une terminologie catégorique : dans cette perspective, il est possible d'énoncer une dualité entre les algèbres —impliquées dans la description des types de données abstraits— et les coalgèbres —qui surgissent dans l'analyse des structures dynamiques état-basées se produisant en informatique. Beaucoup des situations applicables tombent dans le coalgébrique, ou coinductif, secteur : typiquement tous les phénomènes où une quantité d'information infinie est impliquée (par exemple ne terminant jamais les processus, les streams, les nombres réels exacts, les boucles des indicateurs dans la mémoire, et ainsi de suite).

Synthèse. Le présent document est organisé dans trois parts distinctes. Le premier assure le fond utile pour développer les autres deux. Plus avec précision, le chapitre 1 se concentre sur les outils que nous travaillons avec, à savoir les Logical Frameworks basés sur la Théorie de Types, l'aide interactif de preuve Coq et les principes coinductifs. La thèse commence vraiment dans le chapitre 2, où nous rassemblons les motivations pour les travaux que nous menons à bien dans les chapitres suivants.

La deuxième partie de la thèse traite des nombres réels constructifs. Nous construisons les réels dans des le chapitre 3 employant des streams, c.-à-d. ordres infinis, des chiffres signés. Nous travaillons avec la logique constructive, c.-à-d., nous développons des mathématiques "algorithmiques", dans le sens de Bishop [Bis67] et Martin-Löf [ML82]. Nous mettons en application les nombres réels dans Coq comme paires naturels-streams, qui sont contrôlés en utilisant des jugements coinductifs et des algorithmes corécursifs. Dans le chapitre 4 nous présentons une axiomatisation constructive des réels et nous l'employons pour prouver l'adéquation de notre construction. Par conséquent, nous avons obtenu des nombres réels fiables et nous avons certifié les algorithmes (arithmétiques) travaillant sur l'application. D'ailleurs, les fonctions mettant en application les algorithmes peuvent être extraites à partir de Coq, de ce fait fournissant le logiciel certifié.

La troisième partie de la thèse est indépendante de la seconde. Nous approchons le

calculs objet-basés avec des effets secondaires dans le but de l'adressage d'importantes propriétés (meta)theoriques, en tant que par exemple la Solidité de Type. C'est un premier effort vers la synthèse de logiciel certifié pour des langages objet-basés. Nous énoncons notre recherche se concentrant sur le **imp**ς-calcul d'Abadi-Cardelli [AC96], un calcul très représentatif comportant les objets, le clonage, la consultation dynamique, la mise à jour impérative de méthode, les types d'objet et la subsumption. Dans le chapitre 5 nous examinons **imp**ς, qui est après reformulé dans des le chapitre 6 employant des techniques de codage modernes, comme la Syntax Abstraite d'Haut-Ordre (HOAS) et les systèmes coinductifs de preuve dans le modèle de Déduction Naturelle. Ces systèmes de preuve sont employés ici pour la première fois en combination avec HOAS pour traiter un calcul objet-basé et prouver un théorème de Solidité de Type. En conclusion, dans le chapitre 7, nous formalisons **imp**ς dans Coq et nous prouvons formellement la Solidité de Type, profitant pleinement des techniques fournies par $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$, la théorie coinductive de type fondamental de Coq.

Part I

Preliminary reflections

Chapter 1

Formal Methods based on Type Theory

In this chapter we introduce the Logical Frameworks based on Type Theory. We focus in particular on the Calculus of Inductive and CoInductive Constructions $CC^{(Co)Ind}$ [CH88, PM93, Gim94], the coinductive type theory underlying Coq. Then we briefly present the proof assistant, and we make some reflections about coinduction.

1.1 Computational metamodels

Computer Science has grown into a complex discipline, with scientific and technological sides to it, which intersects various knowledge domains at once, and acts at various levels of abstraction.

The semantic and syntactic tools normally used to specify and analyze the object level systems appear on the *first* of these metalevels. This is the abstraction level of programming and specification languages, of calculi, of denotational and operational models, of automata, of Petri Nets, etc., but it is also the level of the logical systems used in verifying and analyzing properties of programs and systems.

The development of Computer Science in the last decades has indicated clearly that there is no chance to come up some day with the universal programming or modeling language, or with the ultimate universal computational logic. We have to live with a plethora of different calculi and special-purpose formalisms, and we have to be constantly ready to develop new conceptual frameworks. But, in this perspective, we do not want to start over from first principles the theory of each and every one of these systems, calculi or logics; nor we want to re-implement from scratch, for each of them, suitable interactive tools for manipulating them rigorously. It is unavoidable therefore to conceive *another* abstraction level above the first one, where commonalities across different systems can be focused and factored out. On this level we can define framework theories within which to prove structural results which prevent us from duplicating efforts in developing the metatheory of the lower level systems. This second metalevel is the level of "computational metamodels". Computational metamodels are those framework theories within which it is possible to develop the metatheory of the calculi, models and logical systems which are currently used to specify and reason on hardware and software systems. Examples of these are Milner's Action Structures, Type Theory-based Logical Frameworks, Rewriting

Logics, Graph Grammars.

Thus a paradigm example of computational metamodel is that of Type Theory-based Logical Frameworks (LFs), as it was introduced and used in [HHP93], building on earlier intuitions of de Bruijn [dB80] and Martin-Löf [NPS90], and developed e.g in [DFH95]. A LF is a specification language for formal systems and their inferential mechanisms, based on constructive type theories such as Martin-Löf Type Theory [NPS90] or the Calculus of Constructions [CH88], possibly extended with mechanisms for (co)inductive definitions [PM93, Gim94]. LFs were introduced with the aim of eliminating useless duplications of particular components in the implementation of interactive environments for assisting in rigorous proof development. In fact, a Logical Framework is a general logic system and any of its implementations provides immediately a generic proof editor. In the last decade many such implementations have been developed: e.g. Coq, Lego, Isabelle, Alf. Logical Frameworks offer a substantial advantage in the specification of syntax and operational semantics of object systems over the more rudimentary BNF: the Higher-Order Abstract Syntax (HOAS). HOAS allows to express in a uniform and algebraic way binding operators over names and variables. And therefore it allows to express context dependent constraints, and provides a standard format for treating α-conversion and scope disciplines. Logical Frameworks, however, allow for/force the user to formulate the systems in a Natural Deduction form (ND) in such a way that the dependency between assumptions and conclusions, i.e. the consequence relation specific to the system, is made explicit. Both HOAS and ND therefore suggest new ways of presenting the object systems, more appropriate for computer assisted formal reasoning.

1.2 Logical frameworks based on typed λ-calculus

Church [Chu33] proposed a general theory of functions and logic as a foundation for mathematics. Even if the whole system was proved to be inconsistent [KR35], the functional part, universally known under the name of λ-*calculus*, became *the* model of functional computation [Chu41]. In the λ-calculus there were no types, i.e., every expression can be applied to every argument (even to itself); whence, it is sometimes called *untyped* λ-calculus.

Indeed, in [Cur34] and [Chu40] two typed versions of the λ-calculus were introduced. In the former document terms are essentially those of untyped λ-calculus; then to every term it is associated a set of possible types (including the empty set), following predefined rules of type assignment. Hence, systems following such a paradigm are also called *type assignment* systems or typed λ-calculi *à la Curry*. On the other hand, the paradigm proposed in [Chu40] features systems known as typed λ-calculi *à la Church*, where terms are annotated with their corresponding type, i.e., they carry type information with them. Moreover, every term has usually a unique type associated with it (while in type assignment systems a term determines a set of possible types).

Type theory-based Logical Frameworks arise from the *Curry-Howard isomorphism*[1] (independently introduced in [dB80] and [How80]), which allows thinking to types not only as partial correctness specifications of programs (terms), but also as *propositions*. Whence, a given term can be interpreted as a proof of the proposition associated with its type; thus a proposition is true (i.e, there is a proof of it) if the corresponding type is inhabited.

[1] Also known as *propositions-as-types* principle.

Moreover, since logical systems can be viewed as calculi for building proofs of a given set of basic judgments, it is possible to state the *judgments-as-types* principle [NPS90, HHP93] for those type theories featuring dependent types. This principle can be regarded as the metatheoretic analogous of the Curry-Howard isomorphism, and allows in to use fruitfully type theories as *general logic programming languages*, i.e. LFs. This fact consists in representing basic judgments with suitable types of the LF, whence proofs are represented by terms whose type represents, in turn, the judgment they prove. Moreover, dependent types allow to extend uniformly basic judgment forms to two higher-order forms introduced by Martin-Löf, namely, the *hypothetical* judgment (representing consequence) and the *schematic* judgment (representing generality). Hence, all the relevant parts of an inference system can be faithfully represented in a LF: syntactic categories, terms, judgments, axiom and rule schemata, and so on.

Pure Type Systems. Typed λ-calculi *à la Church* can be generally described as *Pure Type Systems* (PTSs). Such a formalism emerged from the independent works of Berardi and Terlouw (see [Bar92])[2]. The basic language of a PTS is that of *pseudo-terms*.

Definition 1.1 (Pseudo-terms) *Let V be an infinite set of variable symbols, ranged over by x,y,z, and C an infinite set of constant symbols, ranged over by c. The set of pseudo-terms $\mathcal{T}$, ranged over by M, N, A, B, C, is specified by the following grammar:*

$$\mathcal{T} \qquad M ::= x \mid c \mid MN \mid \lambda x{:}A.M \mid \Pi x{:}A.B$$

where the variable x is bound in $\lambda x{:}A.M$ and $\Pi x{:}A.B$.

Given any pseudo-term M, its set of free variables (denoted by $FV(M)$) is defined as usual, keeping in mind that the only binders are the abstraction operator (λ) and the dependent type constructor (Π).

Definition 1.2 (Pseudo-environments) *Let $\mathcal{T}$ be a set of pseudo-terms:*

- *a* statement *is of the form $M : A$ (where M, $A \in \mathcal{T}$); M is called the* subject *and A the* predicate *of $M : A$;*
- *a* declaration *is of the form $x{:}A$ (where $x \in V$ and $A \in \mathcal{T}$);*
- *a* pseudo-environment Γ *is a finite list of declarations, where all the subjects are distinct; pseudo-environments and their domains (sets of subjects occurring in Γ) are inductively defined by the following rules:*
  - *the empty environment $\langle\rangle$ is a pseudo-environment ($dom(\langle\rangle) = \emptyset$);*
  - *if Γ is a pseudo-environment, x is a variable such that $x \notin dom(\Gamma)$, and A is a pseudo-term, then $\langle \Gamma, x{:}A\rangle$ is a pseudo-environment ($dom(\langle \Gamma, x{:}A\rangle) = dom(\Gamma) \cup \{x\}$).*

In the following we write "$x_1{:}A_1, \ldots, x_n{:}A_n$" instead of "$\langle \ldots \langle\langle\rangle, x_1{:}A_1\rangle, \ldots, x_n{:}A_n\rangle$". Moreover, given two pseudo-environments Γ and Δ, we will denote by $\Gamma \subseteq \Delta$ the fact that each statement $x{:}A$ of Γ is also a statement of Δ. Finally, Γ, Δ stands for the pseudo-environment obtained appending the list of statements of Δ to that of Γ.

[2]Both were approaching the problem of finding a method to generate all the systems of the λ-cube.

1. General axioms and rules:

$$\begin{array}{ll}
\text{(Axiom)} & \langle\rangle \vdash c{:}s \quad (c:s) \in \mathcal{A} \\
\text{(Start rule)} & \dfrac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x{:}A} x \notin \Gamma \\
\text{(Weakening rule)} & \dfrac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma, x : B \vdash M : A} x \notin \Gamma \\
\text{(Application rule)} & \dfrac{\Gamma \vdash M : (\Pi x : A.B) \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]} \\
\text{(Abstraction rule)} & \dfrac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash (\Pi x : A.B) : s}{\Gamma \vdash (\lambda x : A.M) : (\Pi x : A.B)} \\
\text{(Conversion rule)} & \dfrac{\Gamma \vdash M : A \quad \Gamma \vdash B' : s \quad B =_\beta B'}{\Gamma \vdash M : B'}
\end{array}$$

2. Specific rules:

$$(s_1, s_2, s_3) \text{ rule} \quad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x : A.B) : s_3} \quad \text{where } (s_1, s_2, s_3) \in \mathcal{R}$$

Figure 1.1: Typing axioms and rules.

Definition 1.3 *Let $\mathcal{T}$ be a set of pseudo-terms; the* specification *of a PTS is a triple $\langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$, where:*

1. *$\mathcal{S}$ is a subset of C (the elements of S are called* sorts*);*
2. *$\mathcal{A}$ is a set of* axioms *of the form $c : s$ (where $c \in C$ and $s \in \mathcal{S}$);*
3. *$\mathcal{R}$ is a set of* rules *of the form (s_1, s_2, s_3) (where s_1, s_2, $s_3 \in \mathcal{S}$); in the case that $s_2 = s_3$ we simply write (s_1, s_2) instead of (s_1, s_2, s_3).*

As usual, in the following we will denote by $\rightarrow_\beta$ the relation of (one-step) β-reduction and by $=_\beta$ the corresponding equivalence, also known as β-conversion.

The next step is to define the PTS (with β-conversion) determined by a specification $\langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$.

Definition 1.4 *A specification $S = \langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ determines a PTS (with β-conversion, notation $\lambda S = \lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$. The expression on the right is a typed λ-calculus* à la Church *whose typing judgment is a triple $\langle \Gamma, M, A \rangle$, written $\Gamma \vdash M : A$, where Γ is a pseudo-environment and M, A are pseudo-terms. If $\Gamma \vdash M : A$ is derivable by means of the rules in Figure 1.1, then Γ is said to be a (legal) environment and M, A are (legal) terms.*

A PTS is called singly sorted *if:*

1. *$(c{:}s_1)$, $(c{:}s_2) \in \mathcal{A}$ implies $s_1 = s_2$;*
2. *(s_1, s_2, s_3), $(s_1, s_2, s_3') \in \mathcal{R}$, implies $s_3 = s_3'$.*

In the literature there are also Pure Type Systems with $\beta\eta$-conversion, where some care has to be taken in defining the rules of the equality judgment, otherwise one looses the Church-Rosser property (see [HHP93] for example).

Theorem 1.1 (Church-Rosser property) *If $M \to_\beta^* M'$[3] and $M \to_\beta^* M''$, then there exists N such that $M' \to_\beta^* N$ and $M'' \to_\beta^* N$.*

Since their introduction, PTSs have been deeply investigated; for the sake of completeness we recall below some of their main properties (the interested reader is referred to [Bar92, Geu93]).

Free variable lemma. If $\Gamma \vdash M{:}A$, then $FV(M) \cup FV(A) \subseteq dom(\Gamma)$.

Transitivity lemma. If Γ is a legal context such that $\Gamma \vdash x_i{:}A_i$ $(1 \leq i \leq n)$, $\Delta \triangleq x_1{:}A_1, \ldots x_n{:}A_n$ and $\Delta \vdash A{:}B$, then $\Gamma \vdash A{:}B$.

Substitution lemma. If $\Gamma, x{:}A, \Delta \vdash B{:}C$ and $\Gamma \vdash D{:}A$, then $\Gamma, \Delta[D/x] \vdash B[C/x]{:}C[D/x]$.

Thinning lemma. If Γ and Δ are two legal environments such that $\Gamma \subseteq \Delta$ and $\Gamma \vdash A{:}B$, then $\Delta \vdash A{:}B$.

Generation lemma.

1. $\Gamma \vdash c{:}C$ implies that there exists $s \in \mathcal{S}$ such that $C =_\beta s$ and $(c{:}s) \in \mathcal{A}$;
2. $\Gamma \vdash x{:}C$ implies that there exist $s \in \mathcal{S}$ and B such that $B =_\beta C$, $\Gamma \vdash B{:}s$ and $(x{:}B) \in \Gamma$;
3. $\Gamma \vdash (\Pi x{:}A.B){:}C$ implies that there exists $(s_1, s_2, s_3) \in \mathcal{R}$ such that $\Gamma \vdash A{:}s_1$, $\Gamma, x{:}A \vdash B{:}s_2$ and $C =_\beta s_3$;
4. $\Gamma \vdash (\lambda x{:}A.M){:}C$ implies that there exist $s \in \mathcal{S}$ and B such that $\Gamma \vdash (\Pi x{:}A.B){:}s$, $\Gamma, x{:}A \vdash M{:}B$ and $C =_\beta (\Pi x{:}A.B)$;
5. $\Gamma \vdash (MN){:}C$ implies that there exist A and B such that $\Gamma \vdash M{:}(\Pi x{:}A.B)$, $\Gamma \vdash N{:}A$ and $C =_\beta B[N/x]$.

Subject Reduction theorem. If $\Gamma \vdash M{:}A$ and $M \to_\beta^* N$, then $\Gamma \vdash N{:}A$.

Condensing lemma. If $\Gamma, x{:}A, \Delta \vdash M{:}B$ and $x \notin FV(\Delta) \cup FV(M) \cup FV(B)$, then $\Gamma, \Delta \vdash M{:}B$.

Uniqueness of type for singly sorted PTSs. If a PTS is singly sorted, $\Gamma \vdash M{:}A$ and $\Gamma \vdash M{:}B$, then $A =_\beta B$.

We have seen that rules in a PTS specification have the form (s_1, s_2, s_3); in the particular case when $\mathcal{S} = \{*, \Box\}$ ($*$ and $\Box$ are called, respectively, the sort of *terms* and the sort of *types*) and all the rules in $\mathcal{R}$ have the form (s_1, s_2, s_2) (whence, they can be simply written as (s_1, s_2)), we obtain all the systems of the so-called λ-*cube* (see Figure 1.2). These systems differ each other by the choice of rules; possible combinations of rules are listed in Figure 1.3.

[3] We denote by $\to_\beta^*$ the reflexive and transitive closure of $\to_\beta$.

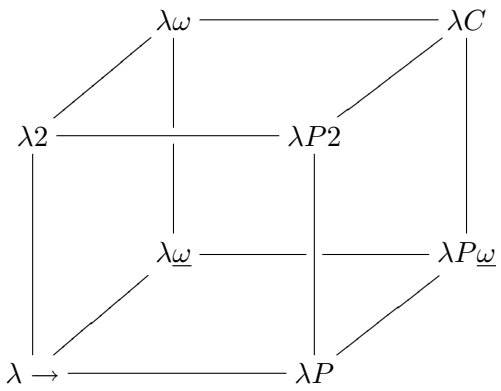

Figure 1.2: The λ-cube.

System	Set of rules			
$\lambda \rightarrow$	$(*,*)$			
$\lambda 2$	$(*,*)$	$(\Box,*)$		
λP	$(*,*)$		$(*,\Box)$	
$\lambda P2$	$(*,*)$	$(\Box,*)$	$(*,\Box)$	
$\lambda\underline{\omega}$	$(*,*)$			$(\Box,\Box)$
$\lambda\omega$	$(*,*)$	$(\Box,*)$		$(\Box,\Box)$
$\lambda P\underline{\omega}$	$(*,*)$		$(*,\Box)$	$(\Box,\Box)$
λC	$(*,*)$	$(\Box,*)$	$(*,\Box)$	$(\Box,\Box)$

Figure 1.3: Rules for the systems of the λ-cube.

As the reader can see, all the systems have the basic rule $(*,*)$, which allows to abstract terms over terms. More sophisticated levels of abstraction can be reached by adding some of the other rules, e.g., all the systems on the upper face of the λ-cube feature the rule $(\Box,*)$, allowing to derive impredicative types, like $\Pi A{:}*.(A \rightarrow A)$[4]:

$$\dfrac{\langle\rangle \vdash *{:}\Box \qquad \dfrac{\dfrac{\langle\rangle \vdash *{:}\Box}{A{:}* \vdash A{:}*} \qquad \dfrac{\langle\rangle \vdash *{:}\Box}{A{:}* \vdash A{:}*}}{A{:}*, x{:}A \vdash A{:}*}}{A{:}* \vdash (A \rightarrow A){:}*}}{\langle\rangle \vdash (\Pi A{:}*.(A \rightarrow A)){:}*}$$

The terminology *impredicative type* is motivated by the following reasons:

1. $\Pi A{:}*.(A \rightarrow A)$, being a Cartesian product of types, is indeed a type $((\Pi A{:}*.(A \rightarrow A)) : *)$;

2. however, since the product is over all the possible types, it includes $\Pi A{:}*.(A \rightarrow A)$, which is in the process of being defined; whence the impredicativity of the definition.

[4]In general, $A \rightarrow B$ is an abbreviation for $\Pi x : A.B$ when x does not occur in B.

The structure of the λ-cube is very useful, since it allows to understand at a glance the expressive power of each PTS labeling its vertices. For instance, the PTS λC, at the top right vertex, is the only one featuring all the possible rules; hence it represents the most general type theory. In the literature λC is also known as the *Calculus of Constructions (CC)* (introduced by Coquand and Huet in [CH88]); other well-known systems related to those of the λ-cube are the simply typed λ-calculus [Chu40] (corresponding to $\lambda \rightarrow$), the *System F* introduced in [Gir72] (corresponding to $\lambda 2$), and the Edinburgh Logical Framework, which corresponds to λP [HHP93].

The most important property enjoyed by the systems in the λ-cube is the *strong normalization*.

Definition 1.5 *Let λS be a PTS. Then λS is* strongly normalizing *($\lambda S \models SN$) if $\Gamma \vdash M{:}A$ implies that both M and A are strongly normalizing ($SN(M)$ and $SN(A)$); i.e., there are no infinite β-reductions starting from M, A.*

Theorem 1.2 (Strong normalization for the λ-cube) *For every system in the λ-cube, the following hold:*

1. *if $\Gamma \vdash M{:}A$, then $SN(M)$ and $SN(A)$;*
2. *$\Gamma \vdash M{:}A$ and $\Gamma \triangleq x_1{:}A_1, \ldots, x_n{:}A_n$ imply $SN(A_i)$ for every i such that $1 \leq i \leq n$.*

The Strong normalization and the Church-Rosser property entail the *decidability of type checking*: so, in order to test whether $U =_\beta V$, reduce both to their unique normal forms, and then check if they are identical up to some renaming of the bound names (i.e., up to α-equivalence). This is the most desirable property of a type theory-based logical framework, since it allows to implement an automated proof-checker. Indeed, by the Curry-Howard isomorphism, proof-checking (that is, verifying that a given proof is indeed a proof of a given proposition) reduces to type checking (that is, verifying that the term corresponding to the given proof inhabits the type corresponding to the given proposition).

1.3 The Calculus of (Co)Inductive Constructions

We take now a closer look to the *Calculus of (Co)Inductive Constructions* ($\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$), an impredicative intuitionistic type theory which extends the original Calculus of Constructions [CH88] with primitive support for inductive [PM93, Wer94] and coinductive types [Gim94].

The $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$ features two basic sorts (i.e. types of types), namely `Prop` and `Set`; the former is intended to be the type of logical propositions, predicates or judgments, while the latter is supposed to be the type of datatypes (e.g., natural numbers, lists, trees, etc.). Since also `Prop` and `Set` can be manipulated as ordinary terms, they must have a type. However, in order to avoid the well known Girard's paradox, it is not possible to assume that these sorts are inhabited by themselves; hence $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$ provides an infinite hierarchy of universes denoted by $\mathtt{Type}(i)$ for any natural i. The hierarchy is such that $\mathtt{Prop}{:}\mathtt{Type}(0)$, $\mathtt{Set}{:}\mathtt{Type}(0)$ and $\mathtt{Type}(i)$ $:\mathtt{Type}(i+1)$. Thus it is possible to state the following PTS-specification for $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$:

- $\mathcal{S} \triangleq \{\mathtt{Prop}, \mathtt{Set}\} \cup \{\mathtt{Type}(i) \mid i \in \mathbb{N}\}$

- $\mathcal{A} \triangleq \{\texttt{Prop} : \texttt{Type}(0), \texttt{Set} : \texttt{Type}(0)\} \cup \{\texttt{Type}(i) : \texttt{Type}(i+1) \mid i \in \mathbb{N}\}$
- $\mathcal{R} \triangleq \{\texttt{Prop}, \texttt{Set}\}^2 \cup \{(\texttt{Type}(i), \texttt{Type}(j), \texttt{Type}(k)) \mid i, j, k \in \mathbb{N}, i, j \leq k\}$

(Co)Inductive definitions. The most appealing feature of $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$ is the possibility of defining types inductively; for instance one can introduce a new inhabitant of `Set` by declaring that it is the least set closed under the application of a given family of constructors. Since inductive definitions can be given by means of higher-order impredicative quantifications in any extension of the (second order) PTS $\lambda 2$, one could conjecture that the inductive types of $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$ are indeed superfluous. However, as remarked in [CP90], impredicative inductive types have two major drawbacks:

1. the impossibility of proving the induction principle;
2. when recursive definitions are called for, the conversion rule for natural numbers yielded by an impredicative definition is $rec(x, f)(S(n)) = f(\phi(n), rec(x, f)(n))$ instead of the expected $rec(x, f)(S(n)) = f(n, rec(x, f)(n))$ (where ϕ is a term such that $\phi(0) = 0$ and $\phi(S(n)) = S(\phi(n))$). Hence, since $\phi(n)$ and n are intensionally equal, but not convertible, programs (terms) containing expressions like $\phi^k(n)$ are inefficient, since the reduction of $\phi^k(S(n))$ to $S(\phi^k(n))$ requires k steps.

Coquand and Paulin-Mohring introduced in [CP90, PM93] a solution for this problem, by extending the language of pseudo-terms of CC with special constants representing the definition, introduction and elimination of inductive types:

$$\mathcal{T} \qquad M ::= \ldots \mid \texttt{Ind}(x{:}A)\{A_1|\ldots|A_n\} \mid \texttt{Constr}(i, A) \mid \texttt{Elim}(M, A)\{M_1|\ldots|M_n\}$$

where i ranges over natural numbers. The informal meaning of the new pseudo-term constructors is the following:

- $\texttt{Ind}(x{:}A)\{A_1|\ldots|A_n\}$ represents an inductive type (denoted by x) of sort A whose constructors are given by $A_1, \ldots, A_n$;
- if $I \triangleq \texttt{Ind}(x{:}A)\{A_1|\ldots|A_n\}$, then $\texttt{Constr}(i, I)$ has type A_i and represents the i-th constructor of the latter;
- $\texttt{Elim}(M, A)\{M_1|\ldots|M_n\}$ is a term defined by induction/recursion over a term M belonging to an inductively defined type, where $M_1, \ldots, M_n$ are the n branches corresponding to the constructors of the latter.

Extending the language with new constructors requires to extend the typing system and the rules stating the equivalence of terms as well. Moreover, a new term reduction rule, called *ι-reduction*, is needed in order to represent the computational content of inductive types (i.e., a way for performing computations with recursively defined functions).

The same approach has been adopted by Giménez in [Gim94] in order to provide a better support for *coinductive types*[5] w.r.t. the classical approach of representing them by means of higher-order impredicative quantifications. Hence, the language of pseudo-terms has been further extended with new constructors[6]:

[5] Coinductive types can be thought of as sets whose elements may be potentially infinite (circular) like, e.g., streams (i.e., infinite sequences) of natural numbers.

[6] The notation used in [Gim94] is slightly different from that in [PM93]; here we prefer to unify them in order to enlighten the duality between inductive and coinductive types.

- `CoInd`$(x{:}A)\{A_1|\ldots|A_n\}$ represents a coinductive type (denoted by x) of sort A whose constructors are given by $A_1, \ldots, A_n$;

- if $I \triangleq$ `CoInd`$(x{:}A)\{A_1|\ldots|A_n\}$, then `Constr`$(i, I)$ has type A_i and represents the i-th constructor of the latter (this is not a new constructor, but simply an extension of `Constr` to coinductive types);

- `CoFix` $f{:}A := M$, where f may occur in M (under certain conditions, briefly discussed below), represents a way of defining a new potentially infinite object, denoted by f, through a "circular" equation. It is important noticing that the aim of the constructor `CoFix` is to generate elements of a coinductive type (and not eliminating them, as the dual constructor `Elim` of inductive types).

Like in the case of inductive types, the new constructors require extensions to the typing system and the equivalence rules. Moreover, ι-reduction must be carefully extended to take into account also a new form of redex, which involves corecursive objects defined by means of the `CoFix` constructor.

The logic underlying the systems of the λ-cube is constructive, whereas infinite objects are non constructive, i.e., they cannot be represented effectively. However, it is sometimes possible to represent the process of generating infinite objects in the case such a process is effective. This is the reason why coinductive types allow to represent potentially infinite objects: in fact, equations defined with the $CoFix$ constructor can be viewed as *generation rules* for infinite terms. Hence, in order to *compute* inhabitants of coinductive types, we should *unfold*[7] such equations, thus yielding some information which can be subsequently used. However, allowing the unfolding of coinductive definitions without restrictions means loosing the Strong normalization property and, consequently, loosing the decidability of type checking. In order to avoid this, Giménez [Gim94] states that unfolding coinductive definitions is possible only in a *lazy* way, i.e., when they are the argument of a *case analysis* constructor.

Since the `CoFix` constructor does not impose any constraint on M in `CoFix` $f{:}A := M$, it would be possible to introduce definitions like `CoFix` $f{:}A := f$, which would yield non-normalizing terms even in the case that only lazy reduction is permitted. In order to avoid this problem, `CoFix`-definitions must satisfy a *guarded-by-constructors* condition[8], which is a *syntactic criterion* derived from the *Guarded Induction Principle* (introduced in [Coq93]). Without entering into the details, the intuition behind such a criterion is that a legal defining equation of a corecursive object must yield some computational content at every reduction step. Thus definitions like `CoFix` $f{:}A := f$ are rejected by the system. This leads to the previously mentioned idea of representing the processes generating infinite objects, provided they are effective.

The extended grammar defining the language of pseudo-terms allows for arbitrary declarations of (co)inductive types. Some of them can easily lead to inconsistencies (i.e., to inhabit the void type[9], corresponding to the absurd). To ensure the soundness of

[7] By unfolding an equation of the form `CoFix` $f{:}A := M$, we mean the operation of substituting it for the free occurrences of f in the body M ($M[(\texttt{CoFix}\ f{:}A := M)/f]$).

[8] Roughly, the *guarded-by-constructors* condition amounts to ensure that the occurrences of f in M must be guarded (i.e., they must be preceded) by a constructor of the corresponding coinductive type.

[9] The void type is defined as $\bot \triangleq$ `Ind`$(x{:}$`Prop`$)\{\}$; the corresponding non-dependent elimination principle is $(P{:}\texttt{Prop}\bot \to P)$ saying that from $\bot$ we can derive any proposition.

recursive type declarations (both inductive and coinductive ones) the constructors must satisfy some conditions, i.e., they must be *forms of constructor* w.r.t. the new defined type. Keeping in mind that in $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$ MN is written $(M\ N)$ and $\Pi x{:}M.N$ is written $(x{:}M)N$, these conditions are formally expressed in the following way (where $\vec{x}{:}\vec{M}$ is an abbreviated notation for $x_1{:}M_1, \ldots, x_n{:}M_n$ and $|\vec{M}|$ stands for the length of $\vec{M}$).

Definition 1.6 *The variable X occurs strictly positively in the term P if $P \equiv (\vec{x}{:}\vec{M})$ $(X\vec{N})$ and X does not occur free in M_i $\forall i \in 1..|\vec{M}|$, nor in N_j $\forall j \in 1..|\vec{N}|$.*

Definition 1.7 *Let X be a variable. The terms which are a form of constructor w.r.t. X are generated by the syntax:*

$$Co ::= (X\ \vec{N}) \mid P \to Co \mid (x{:}M)Co$$

with the further restrictions that X does not occur free in N_i $\forall i \in ..|\vec{N}|$, it is strictly positive in P, and it does not occur free in M.

1.4 The proof assistant Coq

The Coq system [INR03], which is a proof assistant based on $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$, is the result of over ten years of research at INRIA[10].

In 1984, Coquand and Huet wrote the first implementation of the Calculus of Constructions in CAML (a functional language belonging to the ML family and developed at INRIA). The core of the system was a proof-checker, known as *Constructive Engine*, which allowed the declaration of axioms and parameters, the definition of mathematical types and objects and the explicit construction of proof-objects, represented as λ-terms. Then, a section mechanism, called the *Mathematical Vernacular* was added to permit developing mathematical theories in a hierarchical way. At the same time was implemented an interactive *Theorem Prover* for executing tactics written in CAML: it allows to build progressively proof trees in a *top-down* style, thus generating subgoals and backtracking when needed. Moreover, the basic set of tactics could be extended by the user.

With the introduction of inductive types by Paulin-Mohring, was added a new set of tactics permitting to carry out proofs by induction was added. The implementation of a module for compiling programs extracted from proofs in CAML is due to Werner.

Starting from version V5.10, the Coq system supports coinductive types as well, and provides the powerful tactic `Cofix`, which allows to build interactively a proof about coinductive predicates: this is carried out incrementally in a natural way, without having to exhibit a priori a bisimulation, like usually done with "pencil and paper".

The essential features of the Coq system are the following:

1. a logic metalanguage, allowing to represent formal systems;
2. a proof engine, assisting the user in formally reasoning about the encoded systems;
3. a program extractor, yielding functional programs from constructive proofs.

[10] Institut National de Recherche en Informatique et Automatique, France.

Terms, types and sorts. In the previous sections we have seen that type theories allow to manipulate only two categories of objects: terms and types. The latter specify the classes the former can belong to, so every object must belong to a type. In $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$ (the underlying metalanguage of Coq) there are no syntactic distinctions between terms and types, since they can be defined in a mutual way, and similar constructions can be applied to both categories. Hence, also the types of Coq must have a type: this causes the introduction of sorts, which are constants of the system. In section 1.3 we introduced the sorts of $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$, which are infinitely many in order to avoid Girard's paradox. However, in Coq the user does not have to worry about the indexes i of `Type`(i), since the latter are automatically handled by the system. Hence, from the user's viewpoint we have `Prop:Type`, `Set:Type` and `Type:Type`.

The `Gallina` specification language.

We briefly recall the specification language of Coq, called `Gallina`, which allows to develop mathematical theories handling axioms, hypotheses, parameters, constants definitions, functions, predicates and so on. The usual operations of typed λ-calculi are rendered in `Gallina` as follows:

- $\lambda x : M.N$ is written `[x:M]N`,
- (MN) is written `(M N)` (the application is left associative),
- $M \to N$ is written `M -> N`,
- $\Pi x{:}M.N$ is written `(x:M)N`.

Declarations. The system waits for user's commands by means of a prompt (`Coq <`); it is possible to enrich the current environment through the commands `Variable`, `Hypothesis`, `Axiom` and `Parameter`[11] (the use of `Axiom` and `Hypothesis` is recommended for logical entities, while `Variable` and `Parameter` should be used in the remaining cases). For instance, the declaration of a variable ranging over natural numbers (a predefined type in Coq) can be done with the following command:

```
Coq < Variable n:nat.
```

The system communicates that the operation has been carried out successfully by emitting the following message:

```
n is assumed
```

The `Check` command allows to control the type of a previously declared variable:

```
Check n.
```

and in the case of the previous declaration we obtain:

```
n:nat
```

[11] When a section is closed, the objects declared by means of the first two commands are automatically discharged.

Definitions. Definitions differ from declarations in the sense that the former associate a name to a term correctly typable in the current environment, while the latter associate a type to a name. Therefore the name of a defined object can be replaced at any time by the body of the corresponding definition (this is commonly known as δ-reduction). The general form of a definition is the following:

$$\texttt{Definition}\ ident := term.$$

The command `Print`, applied to the name of a defined object, yields the body of the definition and its type.

Inductive types are introduced as follows:

$$\texttt{Inductive}\ ident : term := ident_1 : term_1 \mid \ldots \mid ident_n : term_n.$$

where *ident* is the name of the new object, *term* is its type and the identifiers $ident_1, \ldots,$ $ident_n$ (whose types are, respectively, $term_1, \ldots, term_n$) represent its constructors. Inductive definition must satisfy some syntactic conditions in order to avoid inconsistencies (see Definition 1.7).

For instance, the set of natural numbers can be introduced as follows:

```
Coq < Inductive nat : Set := O : nat | S : nat -> nat.
nat_ind is defined
nat_rec is defined
nat_rect is defined
nat is defined
```

The Coq system automatically generates eliminations principles for the new inductive type: `nat_ind` is associated to the sort `Prop`, `nat_rec` to the sort `Set` and `nat_rect` to the sort `Type`. The first elimination scheme represents the well known *Peano's induction principle*. On the other hand, `nat_rec` encodes primitive recursion on natural numbers.

```
Coq < Check nat_ind.
nat_ind :
   (P:nat->Prop)(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)
```

In a completely analogous way it is possible to define new coinductive types:

$$\texttt{CoInductive}\ ident : term := ident_1 : term_1 \mid \ldots \mid ident_n : term_n.$$

As for inductive definitions the constructors types must satisfy the constraints of Definition 1.7. Obviously, in this, coinductive case, no elimination schemes are yielded by the system.

In both inductive and coinductive definitions it is possible to introduce parametric and mutual types. For further details the reader can refer to [INR03].

Case expressions. A *case expression* is a destructive operation whose underlying idea is taking a term m belonging to a recursive type I (inductive or coinductive) and building a term of type $(P\ m)$ (depending on m) by cases. The syntax is the following:

$$< term > \texttt{Case}\ term\ \texttt{of}\ \underbrace{term_1 \cdots term_n}_{\text{n expressions}}\ \texttt{end}$$

The preceding term, in the case that $m = (c_i\ t_1 \cdots t_p)$, reduces to $(term_i\ t_1 \cdots t_p)$.

`Fixpoint` and `CoFixpoint`. Fixed point definitions on inductive objects can be introduced as follows:

$$\texttt{Fixpoint}\ ident[ident_1 : term_1] : term_2 := term_3.$$

The intuitive semantics is that *ident* represents a recursive function with argument $ident_1$ of type $term_1$ (which has to be inductive) such that ($ident\ ident_1$) is equivalent to $term_3$ of type $term_2$. Hence, the type of *ident* is $(ident_1 : term_1)term_2$. Obviously, fixed point definitions must satisfy some syntactic constraints in order to ensure that the defined function terminates. More precisely, the *structurally smaller calls* principle [Coq93] requires that the recursive calls of the function must apply to *proper* subterms of the recursive argument. Moreover, in order to preserve strong normalization, fixed point definitions can reduce only when the recursive argument is a term beginning with a constructor. Fixed point definitions can also be parametric, mutually defined and defined by *pattern matching* (for more details, the reader can refer to [INR03]).

The `Gallina` specification language also allows to build infinite objects by means of the `CoFixpoint` command, implementing the `CoFix` constructor of $CC^{(Co)Ind}$ introduced in section 1.3 . The syntax is analogous to that of the `Fixpoint` command, while the associated reduction rule is lazily defined in order to preserve the strong normalization property of the system, as we recalled in section 1.3.

The proof editing mode.

When Coq enters the so-called *proof editing mode*, the system prompt changes from `Coq <` to `ident <`, where `ident` is the name of the theorem one wants to prove. Besides the commands briefly recalled so far, other specific commands are available in proof editing mode. At any instant, the proof context is represented by a set of subgoals to prove (initially this set contains only the theorem) and by a set of hypotheses (initially the list is empty) local to any goal. Both sets are manipulated and modified through tactics; when the proof development is completed (i.e., when the set of subgoals is empty), the system notifies the user with the message `Subtree proved!` and the command `Qed` (or `Save`) is made available in order to store the proof in the current environment for later use in subsequent theorems.

The commands starting the proof editing mode are the following:

- `Goal` *term*. In this case we have to prove *term*, whose associated name is `Unnamed_thm`;
- `Theorem` *ident* : *term*. This command has the same effect of `Goal`, with the difference that the current goal is named *ident* (other variants are `Lemma`, `Remark` and `Fact`).

Predefined logic objects. As previously outlined, terms with sort `Prop` represent propositions, while predicates are rendered by means of dependent types and, when applied to the right arguments, yield propositions. In Coq, like in traditional logics, propositions and predicates can be combined together by means of *logic connectives* in order to build other propositions and predicates. The most commonly used connectives are rendered as follows:

- The true constant is denoted by `True` and it is inductively defined by the non-dependent type which is always inhabited:

```
Coq < Inductive True : Prop := I : True.
```

- The false constant is denoted by `False` and is defined by the empty inductive type (i.e., the type with no inhabitants):

```
Coq < Inductive False : Prop := .
```

- The negation of a proposition is equivalent in intuitionistic logic to the possibility of deriving the absurdity from it:

```
Coq < Definition not := [A:Prop] A -> False.
```

- The logic conjunction is denoted by `A /\ B` and is defined by:

```
Coq < Inductive and [A,B:Prop] : Prop
Coq <     := conj : A -> B -> A /\ B.
```

- The logic disjunction is denoted by `A\/B` and is defined by:

```
Coq < Inductive or [A,B:Prop] : Prop
Coq <     := or_introl : A -> A \/ B
Coq <      | or_intror : B -> A \/ B.
```

- The logic implication between propositions is rendered through the non dependent type constructor `->`.

- Universal quantification on a predicate `P` is rendered by means of dependent types. For instance, the formula $\forall x \in A.P(x)$ is encoded by `(x:A)(P x)`.

- Existential quantification is inductively defined as follows:

```
Coq < Inductive ex [A:Set;P:A->Prop] : Prop
Coq <     := ex_intro : (x:A)(P x) -> (ex A P).
```

It is worth noticing that the definitions of implication and universal quantification are a direct consequence of the Curry-Howard isomorphism.

Equality is inductively defined as well:

```
Coq < Inductive eq [A:Set;x:A] : A->Prop
Coq <     := refl_equal : (eq A x x).
```

Usually `(eq ? x y)` is written `x=y` in Coq. The above definition is due to Christine Paulin-Mohring and amounts to the least reflexive relation[12].

[12] It is formally provable that the given definition of equality coincides with Leibniz definition of equality, stating that two object are equal if and only if they satisfy the same properties. Hence, the equality of Coq is often referred to as Leibniz equivalence.

Proof tactics. Generally, an inference rule represents a link between a conclusion and one or more premises: this characterization allows to read it in two different ways. The former amounts to the classical *forward reasoning* and consists of assuming the premises in order to derive the conclusion. The second interpretation is less immediate, but more useful in the field of computer assisted proof development: the so-called *backward reasoning* proceeds from the conclusion to the premises, starting from the idea that it is necessary to derive the premises in order to prove the conclusion. Coq's tactics work in this second, backward way, replacing the current *goal* (the conclusion) with one or more *subgoals* (the premises). So doing, the proof tree is progressively built starting from the root and the subgoals, obtained by the application of a given tactic, represent the roots of the related subtrees. The proof of a subgoal succeeds by means of axioms, hypotheses of the current proof environment or previously proved results.

The Coq system provides a great number of tactics which, can be roughly grouped as follows:

1. Brute force tactics:
   - `Exact` *term*: this tactic can be applied to any goal; if `T` is the current goal and `p` is a term of type `U`, then `Exact p` succeeds if and only if `T` and `U` are convertible types.
2. Basic tactics:
   - `Assumption` is another tactic applicable to any goal: it automatically searches in the current proof environment for an hypothesis whose type coincides with the current goal.
   - `Intro`: it can be applied to any goal having the form of a product; in the case that the latter is a dependent type `(x:T)U`, the effect is to add to the current proof environment the hypothesis `x:T` or `xn:T`, with `xn` fresh in the case `x` is already present. On the other hand, if the goal is a non-dependent product `T->U`, the tactic introduces in the current proof environment an hypothesis of type `T` (if a hypothesis name is not supplied by the user, it is automatically chosen by Coq). The variant `Intros` repeats the tactics `Intro` as much as possible.
   - `Apply` *term*: when applied to any goal, this tactic tries to unify the current goal with the conclusion of the type of *term*, where the latter is a term well formed in the current environment. If the unification succeeds, then the tactic yields as many subgoals as the instantiations of the premises of the type of *term*. An extensively used variant is `Apply` term `with` $term_1 \cdots term_n$, which allows to instantiate all the premises of the type of *term* which are not deducible from the unification.
   - `Cut` *term*: this tactic is very useful in the proof development; it can be applied to any goal and allows to prove the current goal `T` as a consequence of `U`. Thus, `Cut U` replaces `T` with two subgoals, namely, `U->T` and `U`.
3. Introduction tactics:
   - `Constructor i`: applicable when the head of the conclusion of the current goal is an inductive constant `I` (where `i` is a number less or equal to the number of

constructors of `I`). The effect is the same of the sequence `Intros; Apply ci` (where `ci` is the i-th constructor of `I`).

4. Elimination tactics are specialized for proofs by induction and case analysis:

   - `Elim` *term*: it can be applied to any goal under the condition that the type of *term* is an inductive constant; then, depending on the type of the goal, it applies the appropriate destructor. Like in the case of the `Apply` tactic, there is a variant allowing the user to specify the values for the dependent premises of the elimination scheme which cannot be deduced by the matching operation (`Elim` *term* `with` $term_1 \cdots term_n$).
   - `Case` *term*: it can be applied in order to carry out a proof by case analysis; hence, the type of *term* must be inductive or coinductive.

5. Inversion tactics; when the current proof involves (co)inductive predicates, it is very common to fall into one of the following cases:

   - in the proof environment there is an inconsistent instance of a (co)inductive predicate. Hence, the current goal can be proved by absurdity;
   - in the proof environment there is an instance $(I\ \vec{t})$ of a (co)inductive predicate I and we want to derive, for each constructor c_i of I, all the necessary conditions that should hold for $(I\ \vec{t})$ in order it can be proved by c_i.

   Generally, given an instance $(I\ t_1 \cdots t_n)$ of a (co)inductive predicate, the derivation of the necessary conditions such that $(I\ t_1 \cdots t_n)$ holds, is called *inversion*. Inversion tactics can be classified into three categories:

   (a) tactics inverting an instance of a (co)inductive predicate without storing in the current environment the inversion lemma: `Inversion`, `Simple Inversion` and `Inversion_clear`;

   (b) tactics generating and storing in the current environment the inversion lemma used to invert an instance of a (co)inductive predicate: `Derive Inversion` and `Derive Inversion_clear`;

   (c) tactics inverting an instance of a (co)inductive predicate using an already defined inversion lemma: `Use Inversion`.

6. Conversion tactics:

   - `Change` *term*: it can be applied to any goal; `Change U` replaces the current goal `T` with `U` if and only if `U` is legal and `T` and `U` are convertible.
   - `Simpl`: if `T` is the current goal, this tactic first applies the $\beta\iota$-reductions, then unfolds transparent constants[13] and finally applies again $\beta\iota$-reductions until possible.
   - `Unfold` *ident*: this tactic replaces in the current goal all the occurrences of the transparent constant *ident* with the body of the corresponding definition (this process is also called δ-reduction).

[13] Constants become non-transparent by applying the `Opaque` command.

7. Automatic tactics:

   - `Auto`: this tactic uses a *Prolog-like resolution*, in the sense that it first tries the tactic `Assumption`. If this fails, the tactic `Intro` is applied until the current goal is an atomic one (adding the generated hypotheses as hints); then it tries each of the tactics associated to the head symbol of the goal starting from those with lower costs. The whole process is recursively applied to the generated subgoals.
   - `Trivial`: this is a variant of `Auto` which is not recursive and tries only hints whose cost is zero (see below the `Hint` command).

   `Auto` and `Trivial` use *hints*, organized in several databases, in order to automatically solve the current goal. Each database maps *head symbols* to *hints list* (consisting in a collection of tactics): each hint has a cost[14] and a pattern. `Auto` uses it if the conclusion of the current goal matches its pattern. In the case that several distinct hints are applicable at the same time, the ones with lower costs are tried first. Such a list can be extended by the user using the command `Hint`.

Obviously, we have not covered the whole set of tactics provided by Coq, but we hope to have given an idea of how the proof development with this system can be carried out. The interested reader is referred to [INR03].

1.5 Circularity and Coinduction

It is well-known that structural induction fails on circular, not well-founded objects, thus in recent years considerable effort has been devoted towards the development of simple principles and techniques for understanding, defining and reasoning on this kind of objects. A natural setting is given by those frameworks providing the use of coinductive definitions and proof principles. We briefly present set-theoretical and categorical descriptions of coinduction. After we reconsider Type Theory.

Set-theoretical coinduction.

The recursion theorem due to Tarski [Tar55] states a criterion to guarantee whether a normal equation (described by an endo-operator on a domain) has fixed point solutions.

Theorem 1.3 ***(Fixpoint Theorem - Tarski)***
Let $(L, \leq)$ be a complete lattice and $H : L \to L$ a monotone operator.
Then H has a complete lattice of fixed points; in particular, if μH and νH denote the least and greatest fixed points of H, the following equalities hold:

$$\mu H = \bigcap \{x \in L.\ H(x) \leq x\}$$

$$\nu H = \bigcup \{x \in L.\ x \leq H(x)\}$$

□

[14] Given by the number of subgoals generated by the corresponding tactic.

The above theorem is a good starting point which leads to the concepts of (co)induction and (co)recursion; it actually allows to establish the following chain of conceptual proportions:

$$\frac{\text{Least fixed point}}{\text{Greatest fixed point}} = \frac{\text{Induction}}{\text{Coinduction}} = \frac{\text{Recursion}}{\text{Corecursion}} = \frac{\text{Inductive types}}{\text{Coinductive types}}$$

A basic coinduction principle arises naturally from the above characterization of greatest fixed points: it has been originally used by Milner [Mil83] for reasoning on CCS processes.

Theorem 1.4 ***(Coinduction Principle - Tarski)***
Let $\Phi : \mathcal{P}(X \times Y) \to \mathcal{P}(X \times Y)$ be a monotone operator over the complete lattice of binary relations on the Cartesian product of the sets X and Y, and $\approx_\Phi$ the greatest fixed point of Φ. The following principle is sound:

$$\frac{x \mathcal{R} y \quad \mathcal{R} \subseteq \Phi(\mathcal{R})}{x \approx_\Phi y}$$

The Coinduction Principle is also complete, in the sense that:

$$x \approx_\Phi y \implies \exists \mathcal{R} \,.\, (x \mathcal{R} y \wedge \mathcal{R} \subseteq \Phi(\mathcal{R}))$$

Note that a Φ-bisimulation is a relation $\mathcal{R}$ satisfying the second premise of the principle. □

Starting from this basic schema it is possible to devise suitable generalizations, which are obtained looking for alternative characterizations of greatest fixed points, possibly exploiting special properties of the operator Φ, which allow to relax the condition $\mathcal{R} \subseteq \Phi(\mathcal{R})$. Natural examples [San95, Len98] are inspired to Milner's bisimulation "up-to" notion.

Even if it would be possible to justify definitions given by corecursion and coiteration by means of greatest fixed points —a purely set-theoretic treatment of corecursion and coiteration is described in [Gim94, Geu92]— we prefer to discuss these entities in a categorical setting, which seems more natural.

Category-theoretical coinduction and coalgebras.

A more general approach to coinduction can be stated through categories. The categorical explanation of coinduction, undertaken by Aczel [Acz88], is suggestive and uniform, thus permitting to relate induction and coinduction by means of duality.

The categorical description of coinduction is based on the notion of coalgebra. The distinction between algebra and coalgebra pervades computer science, and has been recognized by many scientists in many situations: essentially this dualism can be understood as the theory of abstract data types versus that of state-based dynamical systems, i.e. construction versus observation. A simple sketch of the arising notions is the following:

$$\frac{\text{Functor}}{\text{Functor}} = \frac{\text{Algebra}}{\text{Coalgebra}} = \frac{\text{Initial algebra}}{\text{Final coalgebra}}$$

We are introducing the necessary definitions only assuming a little elementary theory and simply living in the universe given by the category of sets and functions.

Definition 1.8 ***(Coalgebras, homomorphisms, final coalgebras)*** *Given a functor F, an F-coalgebra is a pair (C, c) consisting of a set C (the carrier, or the set space) and a function $c : C \to F(C)$ (the operation).*

An homomorphism (or a map) from an F-coalgebra (C_1, c_1) to an F-coalgebra (C_2, c_2) consists of a function $m : C_1 \to C_2$ between the carrier sets, which commutes with the operations —i.e. $(c_2 \circ m = F(m) \circ c_1)$, as expressed by the diagram:

$$\begin{array}{ccc} C_1 & \xrightarrow{m} & C_2 \\ {\scriptstyle c_1}\downarrow & & \downarrow{\scriptstyle c_2} \\ F(C_1) & \xrightarrow[F(m)]{} & F(C_2) \end{array}$$

A final F-coalgebra (T, t) is an F-coalgebra such that for every F-coalgebra (C, c) there exists a unique homomorphism of coalgebras $(C, c) \to (T, t)$. □

What is typical of coalgebras is that they can be thought as black-box structures made of state spaces. In particular, no operations for constructing elements are available, thus seeing coalgebras as models of a signature of destruction/observation operations. Among coalgebras, the final ones enjoy a relevant role: it is immediate to associate them a definition and a proof principle which corresponds directly to existence and uniqueness of homomorphisms considered in the above definition.

We are interested in explaining these and other notions referring to a particular (final) coalgebra: the set of infinite sequences, i.e. streams, of elements of some parametric set.

Streams. Let us consider the functor $T(X) = \mathbb{N} \times X$, where $\mathbb{N}$ is the set of natural numbers. A T-coalgebra (S, s) consists of two functions $\texttt{value} : S \to \mathbb{N}$ and $\texttt{next} : S \to S$. Given an element $s \in S$, using these operations we can respectively produce an element of $\mathbb{N}$ ($\texttt{value}(s)$) and a next element of S ($\texttt{next}(s)$). Successively, we can combine the two operations forming another element of $\mathbb{N}$, namely $\texttt{value}(\texttt{next}(s))$; proceeding further we can get, for each element $s \in S$, an infinite sequence $\{s_0, s_1, s_2, \ldots\} \in \mathbb{N}^{\mathbb{N}}$ of elements $s_i = \texttt{value}(\texttt{next}^{(i)}(s))$, $i \in \mathbb{N}$. Such a sequence that s gives rise to is what we can observe about s: in particular, two elements $s, t \in S$ may give rise to the same sequence, thus being observationally undistinguished, i.e. bisimilar.

The next step is to claim that the final coalgebra of the functor T is the set of streams of natural numbers $\mathbb{N}^{\mathbb{N}}$, whose coalgebra structure is:

$$\langle \texttt{head}, \texttt{tail} \rangle : \mathbb{N}^{\mathbb{N}} \to \mathbb{N} \times \mathbb{N}^{\mathbb{N}}$$

defined by:

$$\texttt{head}(\alpha) = \alpha(0) \qquad \texttt{tail}(\alpha) = \lambda x.\ \alpha(x+1)$$

If we consider an arbitrary T-coalgebra $\langle \texttt{value}, \texttt{next} \rangle : S \to \mathbb{N} \times S$, there is a unique homomorphism of coalgebras $h : S \to \mathbb{N}^{\mathbb{N}}$, given, for $s \in S$ and $n \in \mathbb{N}$, by:

$$h(s)(n) = \texttt{value}(\texttt{next}^{(n)}(s))$$

It follows that $(\texttt{head} \circ h = \texttt{value})$ and $(\texttt{tail} \circ h = h \circ \texttt{next})$, thus making h the unique map (up-to-isomorphism) from S to $\mathbb{N}^{\mathbb{N}}$.

Now we can use the finality of $\mathbb{N}^{\mathbb{N}}$ for defining corecursive functions into it; moreover, we can exploit the same finality for building coinductive proofs.

Functions and proofs on streams. We define coinductively the function $\texttt{from} : \mathbb{N} \to \mathbb{N}^{\mathbb{N}}$, which maps a natural number $n \in \mathbb{N}$ to the sequence $(n, n+1, n+2, \ldots) \in \mathbb{N}^{\mathbb{N}}$. We exploit the existence aspect of the finality of $\mathbb{N}^{\mathbb{N}}$ defining the function `from` as the unique function satisfying the following diagram:

$$\begin{array}{ccc} \mathbb{N} & \xrightarrow{\texttt{from}} & \mathbb{N}^{\mathbb{N}} \\ {\scriptstyle \lambda n.(n,n+1)}\downarrow & & \downarrow{\scriptstyle \langle \texttt{head},\texttt{tail}\rangle} \\ \mathbb{N} \times \mathbb{N} & \xrightarrow[\texttt{id}_{\mathbb{N}} \times \texttt{from}]{} & \mathbb{N} \times \mathbb{N}^{\mathbb{N}} \end{array}$$

Introducing in the same way the well-known functions $\texttt{odd} : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}^{\mathbb{N}}$, $\texttt{even} : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}^{\mathbb{N}}$ and $\texttt{merge} : \mathbb{N}^{\mathbb{N}} \times \mathbb{N}^{\mathbb{N}} \to \mathbb{N}^{\mathbb{N}}$, we can prove by finality that:

$$\forall\, \alpha \in \mathbb{N}^{\mathbb{N}}.\ \texttt{merge}(\texttt{odd}(\alpha), \texttt{even}(\alpha)) \cong \alpha$$

where $\cong$ denotes the coinductive, pointwise observational equality. This result follows from uniqueness, because the identity function $\texttt{id}_{\mathbb{N}^{\mathbb{N}}} : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}^{\mathbb{N}}$ is trivially an homomorphism $(\mathbb{N}^{\mathbb{N}}, \langle \texttt{head}, \texttt{tail}\rangle) \to (\mathbb{N}^{\mathbb{N}}, \langle \texttt{head}, \texttt{tail}\rangle)$. It suffices to show that $(\texttt{merge} \circ \langle \texttt{odd}, \texttt{even}\rangle)$ is an homomorphism $(\mathbb{N}^{\mathbb{N}}, \langle \texttt{head}, \texttt{tail}\rangle) \to (\mathbb{N}^{\mathbb{N}}, \langle \texttt{head}, \texttt{tail}\rangle)$ too, thus concluding $(\texttt{merge} \circ \langle \texttt{odd}, \texttt{even}\rangle) = \texttt{id}_{\mathbb{N}^{\mathbb{N}}}$. Therefore, the full proof reduces to show that the following diagram is commuting (a task exploiting the definitions of the functions `odd` and `even`):

$$\begin{array}{ccc} \mathbb{N}^{\mathbb{N}} & \xrightarrow{\texttt{merge} \circ \langle \texttt{odd},\texttt{even}\rangle} & \mathbb{N}^{\mathbb{N}} \\ {\scriptstyle \langle \texttt{head},\texttt{tail}\rangle}\downarrow & & \downarrow{\scriptstyle \langle \texttt{head},\texttt{tail}\rangle} \\ \mathbb{N} \times \mathbb{N}^{\mathbb{N}} & \xrightarrow[\texttt{id}_{\mathbb{N}} \times \texttt{merge} \circ \langle \texttt{odd},\texttt{even}\rangle]{} & \mathbb{N} \times \mathbb{N}^{\mathbb{N}} \end{array}$$

We have seen how coinductive definitions and proofs are obtained through (existing and unique) homomorphisms. An alternative terminology for these entities is coiterative morphisms: an advantage of approaching coinduction with categories is actually that it is possible to deal uniformly with coinductively defined objects and coiterative morphisms, which are used for defining corecursive functions.

An important purpose pointed out in the literature is to generalize the pure coiterative schema, in order to capture a larger class of corecursive functions [Geu92, Len99]. Other interesting investigations concern the study of the relationship between set-theoretic and categorical descriptions of coinduction. We can say here that even if Tarski's coinduction principle can be generated from a suitable coalgebraic coinduction principle based on F-bisimulations [Rut98] (the categorical counterparts of set-theoretic bisimulations), there exist anyway a wide range of contexts where set-theoretic coinductive entities have not yet been explained coalgebraically [Len99].

Type-theoretical coinduction.

As already mentioned in section 1.4, the Coq system allows to define infinite objects like, e.g., *streams* of elements of a parameter type A. When reasoning about coinductive types, the familiar induction principle is not applicable, since constructors, differently from the inductive case, can be applied infinitely many times. On the other hand, case analysis

always yields finite proofs, being practically useless when one must prove properties involving infinite objects. The Guarded Induction Principle [Coq93, Gim94] overcomes these difficulties allowing to apply as an additional hypothesis the property one is proving, under the condition that such hypothesis is used in a way controlled by the constructors of the coinductive type involved.

In parts II and III of the thesis we deal extensively with these coinductive proofs in Coq. Since the aforementioned principle can appear at a first sight a weird thing, in section 3.4 we illustrate its expressive power by means of a simple example.

Chapter 2

Focusing on Real Numbers and Objects

We collect and discuss in this chapter the main motivations for addressing the formal study of (computer programs for) real number computations and object-based languages. Sections from 2.1 to 2.3 concern real numbers; the remaining ones deal with objects.

2.1 Motivations for Real Numbers

Computer programs for high-precision numerical applications are largely used in practice —as for example in numerical analysis, computational geometry, hybrid systems, and so on— but seldom their reliability is addressed formally. A rigorous approach to this kind of software is nowadays crucial for many disciplines, as physics, engineering, aeronautics, and others, because they employ extensively computer systems for performing scientific, numerical elaboration. A first step towards the development of dependable information technology for these disciplines is the implementation of reliable real numbers.

Real numbers have been studied deeply in Mathematics along the centuries. Conversely, in Computer Science, which is considerably younger, the management of these entities is greatly influenced by technology. Indeed, the "real" datatype is embarked nowadays in *programming languages* and *computer algebra* systems, but the common approach to real number computation is not dependable.

Programming languages typically provide *floating-point* real numbers, namely approximations of reals by means of rational numbers, but this kind of "machine" numbers fail to form a field, already by failing to be closed under the field operations.

Computer algebra systems (as `Maple`, or `Mathematica`) perform symbolic computations like factorizing polynomials, solving equations and expanding functions in power series: that is, routine activities that are time-consuming and error-prone for humans. The reals are either still represented by floating-point arbitrary precision techniques, or by means of high-precision formal calculi, that is, formalisms carrying out symbolic manipulations.

Programming languages and computer algebra systems may be powerful and fast, but it is still possible to exhibit counterexamples enlightening their unreliability w.r.t. the pragmatic of computing on the real numbers [JMMZ01]. The crucial point is that in these environments it is not possible to address the *formal verification* of software. Therefore we ask for a different technology, providing this lacking ingredient.

A rigorous approach would require an environment that allows, besides specification and representation features, the possibility of carrying out *mathematical proofs*. These proofs would concern all kinds of objects we can represent in the system, namely specifications, behaviors, datatypes, and thus also real numbers. We find this kind of functionality in type theory-based logical frameworks (LFs) and their implementations, i.e. proof assistants and theorem provers.

Typically, LFs supply as primitive only *discrete* numerical datatypes, as naturals, integers and eventually rational numbers. But this, obviously, does not suffice for addressing the formal study of many systems where a continuous mathematical infrastructure is involved.

Hence our goal is to *implement* the real numbers in logical frameworks. In this respect, we can immediately distinguish possible alternatives: we could axiomatize the reals or to construct them; and we have to choose the logic setting for working with the reals.

An *axiomatic* approach is fast, because it allows to undertake immediately the development of the Analysis; but it poses the problem of justifying formally the axioms (non contradiction, perspicuity, completeness, categoricity) and, especially, it does not permit to *implement* algorithms on the reals.

On the contrary, the *construction* of the reals is a non trivial effort, but it allows for a direct approach to the certification of both the implementation and the algorithms which work on it. The best known techniques for constructing the reals are Cauchy sequences (Cantor's method), Dedekind cuts and positional expansions.

Goal of the research. We choose to *construct* the reals, because we are interested in certifying algorithms and synthesizing reliable software. Correspondingly, we have to choose a suitable logic setting. The use of *classical* logic is well-suited for developing the classical analysis, which is consolidated. The use of *intuitionistic*, or *constructive*, logic is quite well-suited with respect to the *computational* perspective.

We intend to adhere, in our work, to the constructive setting à la Brouwer [Bro07] and Bishop [Bis67]: constructive mathematics is *algorithmic* mathematics, i.e. *"mathematics which uses intuitionistic logic"*. Therefore constructive mathematics is a creative activity [BR99], where the phrase "there exists" should be interpreted strictly as "there can be constructed", or, even better, "we can compute". This point of view has an immediate counterpart in computer science: in fact, Martin-Löf develops his *constructive type theory* [NPS90] pointing out the similarity between mathematics and programming: *"computer science is equivalent to completely formalized mathematics that uses only intuitionistic logic."*

Next we have to choose a suitable representation for the reals. It is immediate that floating-point numerical calculation is not reliable. A dependable alternative is *exact (real number) computation*. Exact computation, which originates from seminal ideas of Brouwer in 1920 and Turing in 1937, is gaining continuously growing interest in recent years [PEE97, Sim98, Wei00]. It is one of the possible solutions which have been introduced to avoid completely the round-off practice, thus permitting to obtain correct results of desired precision from computations, without having to carry out any independent error analysis. Among the several alternatives for representing the reals, we prefer *digit expansions*. This choice is motivated by the fact that it avoids the preliminary step of constructing the rational numbers, and because this solution is simple and naturally close

to the computational perspective.

Finally, we claim that our objective is to provide LFs with an exact implementation of the real numbers, using infinite digit expansions, and such that the reals are a concrete structure suitable both for reasoning and for calculating. This is a direct effort towards computer assisted formal reasoning on reals, and it should devise a workbench for specifying and certifying algorithms on the reals and thus for developing reliable software for numerical applications. We are going to spend the next three sections for outlining the aspects discussed so far.

2.2 Constructions of the Real Numbers

In classical mathematics there exist well-established methods for constructing the real numbers out of something simpler (natural numbers, integers or rationals). The best known approaches are:

- Cauchy sequences;
- Dedekind cuts;
- positional expansions.

Notice, however, that there are plenty of alternative methods, e.g. continued fractions, or a technique (due to Bolzano) based on decreasing nested intervals. A more radical approach is given by Conway [Con76]. There are also some interesting methods based on the "point free topology" construction given by Johnstone [Joh82]. A detailed development which uses the idea of an intuitionistic formal space is given by Negri and Soravia [NS95], especially interesting for constructivists.

Cantor's method. The method generally attributed to Cantor identifies a real number with the set of all Cauchy sequences $(s_n)_{n\in\mathbb{N}}$ that converge to it:

$$\forall \epsilon > 0.\ \exists N \in \mathbb{N}.\ \forall m, n \geq N.\ |\, s_m - s_n \,| < \epsilon$$

The real numbers are defined as the equivalence classes of the following equivalence relation on Cauchy sequences. Given $(s_n)_{n\in\mathbb{N}}$ and $(t_n)_{n\in\mathbb{N}}$:

$$\forall \epsilon > 0.\ \exists N \in \mathbb{N}.\ \forall n \geq N.\ |\, s_n - t_n \,| < \epsilon$$

The arithmetic operations are inherited from those of the rationals in a natural way (for example: $(x+y)_n \triangleq x_n + y_n$), whereas the supremum presents slightly more difficulty.

Dedekind's cuts. A method due to Dedekind identifies a real number with the set of the rationals less than it, which is called *cut*. The properties requires by a set C to be a cut are the following:

- $\exists x.x \in C$;
- $\exists x.x \notin C$;
- $\forall x \in C.\ \forall y < x.\ y \in C$;

- $\forall x \in C.\ \exists y > x.\ y \in C$.

These points state, respectively, that a cut is not empty, is not $\mathbf{Q}$ in its entirety, is downward closed and has no greatest element. As for Cantor's method, the arithmetic operations are inherited from $\mathbf{Q}$ (but the multiplication is problematical), and the supremum of a set of cuts is simply its union.

Positional expansions. Perhaps the most obvious approach is to model the real numbers by infinite positional sequences. It is necessary taking care that the representation is not unique (for example, in standard binary notation $0.111\ldots$ and $1.000\ldots$ represent the same number). Thus one takes equivalence classes and defines easily the ordering, whereas the arithmetic operations are harder to deal with. See section 2.3 for deeper considerations about positional representations.

2.3 Exact computation

In the usual approach to real number computation, one uses a large, but finite. set $M \subset \mathbb{R}$ of "machine numbers": these are typically *floating-point* numbers. The main problem with this approach is that M fails to form a field, already by failing to be closed under the field operations. The literature reports plenty motivating examples where the floating-point approach via rational numbers causes serious consequences, as unreliability, economic waste and even loss of life. Consider for example series not anymore converging [JMMZ01], problems in computational geometry [EH02], or the two disasters of Patriot Missile failure in 1991 Gulf War and explosion of Arian 5 in 1996[1].

Several solutions have been proposed in order to overcome the problem outlined above:

- interval arithmetic: sometimes effective (the intervals may grow very large);
- stochastic arithmetic: sometimes effective, but only probabilistically reliable;
- multiple-precision arithmetic (libraries, `Maple`, `Mathematica`): more effective and reliable, but still uneffective and unreliable;
- exact arithmetic, which we adopt in the present thesis.

The most obvious choice for developing exact arithmetic is to represent the real numbers by *potentially infinite* sequences of digits, which have to be managed starting from the *most significant* digits. However, this naive solution is not adequate; consider for example:

$$3 \cdot (0.333\ldots)$$

It turns out that we are not able to produce the first output digit looking at a finite prefix of $0.\overline{3}$. In fact, if we produced 1 after looking the first n digits, we would do the same also for:

$$3 \cdot (0.\underbrace{3\cdots3}_{\text{n}}2\ldots)$$

which is wrong. Similar arguments apply if we produced 0 as first output digit.

There are plenty of alternative solutions for this computational problem, which are all equivalent:

[1] See `http://www.ima.umn.edu/~arnold.f96/disasters.html`

- any integral base with negative digits (Leslie 1817, Cauchy 1840);
- base 2/3 with binary digits (Brouwer 1920, Turing 1937);
- nested sequences of rationals intervals (Grzegorczyk 1957);
- Cauchy sequences of rationals with fixed rate of convergence (Bishop 1967);
- continued fractions (Vuillemin 1988);
- base golden-ratio with binary digits (Di Gianantonio 1996);
- infinitely iterated Möbius transformations (Edalat and Potts 1997).

In chapters 3 and 4 we choose a signed-digit binary notation for representing exact real numbers and implementing exact algorithms on them.

It is worth noticing that many non-commercial packages for exact real number computation have been implemented in a variety of programming languages. Exact arithmetic is reliable and often effective, but less efficient than floating-point practice. Probably, the best solution would be the combination of computer algebra systems with engines for exact numerical evaluation of symbolic expressions.

2.4 Constructive Mathematics, Computer Science

The history of modern constructive mathematics began with the publication of Brouwer's doctoral thesis [Bro07], in which he gave the first exposition of his philosophy of intuitionism (a general philosophy, not merely one for mathematics). According to Brouwer, the phrase "there exists" should be strictly interpreted as "there can be constructed" or, in modern terminology, "we can compute". In turn, this interpretation of existence led Brouwer to reject the unbridled use of the *Law of Excluded Middle*: $P \vee \neg P$, in mathematical arguments. This rejection has some immediate consequences at the level of the real number line $\mathbb{R}$. Namely, all the following propositions cannot be proved constructively:

- $\forall x \in \mathbb{R}.\ (x = 0) \vee (x \neq 0)$;
- $\forall x \in \mathbb{R}.\ (x < 0) \vee (x \neq 0) \vee (x > 0)$ (Law of Trichotomy);
- each not empty subset of $\mathbb{R}$ that is bounded above has a least upper bound (Least-upper-bound Principle);
- every real number is either rational or irrational.

Further developments of constructive mathematics are due to Heyting, a former student of Brouwer, who in 1930 published axioms for the intuitionistic propositional and predicate logic. And to Markov, who, from the 1940s, practiced recursive mathematics using intuitionistic logic. However, by the mid-1960s it appeared that constructive mathematics was at best a minor activity.

The situation changed dramatically with the publication of Bishop's *Foundations of Constructive Analysis* [Bis67], which led to a renewed interest in constructive mathematics, especially among logicians and computer scientists. Bishop's development was based on a primitive, unspecified notion of *algorithm*, or "finite routine". Bishop's algorithmic

mathematics appears to be equivalent to "mathematics that uses only intuitionistic logic". Although he has been criticized for this lack of precision about the notion of algorithm, it is precisely that "defect" that allows it to be interpreted in a variety of models.

The first explicit use of intuitionistic logic in connection with computer science was in the paper *Constructive Mathematics and Computer Programming* [ML82]. In a series of papers, Martin-Löf develops the philosophical and formal basis for his constructive set theory, or *constructive type theory*, and then points out and exploits the similarity between mathematics and programming. For him algorithmic mathematics, that is, computer science, is equivalent to "completely formalized mathematics that uses only intuitionistic logic".

2.5 Motivations for Objects

Several forms of *object-oriented* programming languages exist. Usually, a classification criterium is to distinguish between class-based and object-based ones.

Class-based languages, as Simula, SmallTalk, C^{++} and Java, form the mainstream of the object-orientation. These languages are centered around the notion of class as description of the structure of objects: each object is generated by a class, and inheritance is determined by the class.

Object-based languages, as Emerald, Cecil, Omega, Obliq and Self, have emerged gradually as simpler and more flexible solutions. They are characterized by the absence of classes and by constructs for the creation of individual objects, which may be generated from other objects, inheriting their properties.

Despite the apparent success and enormous spread in computer applications, the object-oriented paradigm remains an area in which relatively little formal work has been done. Our thesis tries to go in this direction, using formal methods based on type theory for reasoning formally on the semantics of object-based languages.

Research lines. The third part of the thesis (starting with chapter 5) is carried out in the framework of the INRIA Project Miró[2], which is a very large research effort involving, among other, the study, definition and certified implementation of a typed class-based language (of the SmallTalk family) and its intermediate object-based language (of the Self family). The core objective is that the class-based language, SmallTalk2K, keeps the same expressivity of languages as SmallTalk, Self, Beta and, on the other hand, is equipped with a typing discipline guaranteeing a static detection of *message-not-found* potential runtime errors. A crucial point of the research is the study and design of the intermediate language FunTalk, which is object-based, and is the target of the pre-compilation of SmallTalk2K source language.

The aims of the Miró project concerning FunTalk are quite ambitious with respect to the ones of this thesis. The final goal is to obtain certified tools for it (i.e. interpreters, type-checkers, compilers, etc.) using the proof assistant Coq. In the present document we have undertaken these advanced objectives of the project.

We have devised that Abadi-Cardelli's **imp**ς-calculus [AC96] is sufficiently expressive to be considered the kernel of real programming languages, as for example Obliq [Car95],

[2] "Miró: Systèmes à objets, Types and Prototypes: Sémantique et Validation", INRIA-Lorraine and INRIA-Sophia Antipolis

and thus also FunTalk. Hence we have focused our investigation exactly on **imp**ς: the aim of our work is to represent **imp**ς in Coq and to develop its metatheory; we fix as ultimate goal the proof of the Type Soundness for the typing discipline.

2.6 Formal Methods and Objects

The formalism of coalgebras is particularly well suited to describe classes of the object-oriented paradigm [Jac96]: coalgebras can be understood as implementation-independent semantics of state-based dynamical systems, which can be seen as black boxes providing an interface towards the external world [JR97].

The coalgebraic approach to class-based programming certification, undertaken in the seminal work [Rei95], has been considerably extended by a joint effort between the Universities of Dresden and Nijmegen. Such a research line has lead to the development of a very extensive project concerning the specification and verification of object oriented specifications and programs. Essential steps have been the synthesis of the language CCSL [RTJ00] —suitable for describing class specifications— and the front-end compiler LOOP, which translates CCSL specifications into ad hoc representations in the proof assistants PVS or Isabelle. These systems are used for formally reasoning about typical concepts as bisimulations, invariants and behavioral equivalences. Such a technique has been applied to real programs in C^{++} [Tew00], Java [J$^+$98, JHHT98] and JavaCard [vdBJP01]. In these case studies, proof methods tailored to the particular languages have been developed, e.g. based on Hoare logic [Hui01].

In spite of this large application of theoretical techniques to class-based languages, relatively little or no formal work has been done on object-based languages. Although these languages are less used in practice than the class-based ones, they are appreciated for the simplicity and the availability of more primitive mechanisms. Class notions can be emulated by object-based languages in a very flexible way: for example classes themselves can be modeled through objects able to receive the message `new`, which corresponds to the creation of another object. Thus object-based languages can be seen as target languages for a mathematical analysis of the languages based on classes.

From a foundational point of view, indeed, most of the *calculi* introduced for the mathematical analysis of the object-oriented paradigm are object-based, as e.g. [FHM94, AC96]. Among the several calculi, Abadi and Cardelli's **imp**ς [AC96] is particularly representative: it features objects, cloning, dynamic lookup, method update, types, subtypes, and imperative features. This makes **imp**ς quite complex, both at the syntactic and at the semantic level: **imp**ς features all the idiosyncrasies of functional languages with imperative features; moreover, the store model underlying the language allows for loops, thus making the typing system quite awkward. This level of complexity is reflected in developing metatheoretic properties: for instance, the fundamental *subject reduction* and *type soundness* are much harder to state and prove for **imp**ς than in the case of traditional functional languages.

It is clear that this situation can benefit from the use of proof assistants, as Coq, where the theory of the object calculus can be formally represented in some metalanguage, the proofs can be checked and new, error-free proofs can be safely developed in interactive sessions. However, up to our knowledge, there is no formalization of an object-based calculus with side-effects like **imp**ς, yet.

Therefore, the aim of this thesis is to represent, understand and reason on $\mathbf{imp}\varsigma$ and its theory, using the proof assistant Coq and taking most advantage of the features of the underlying coinductive type theory $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$.

Encoding methodology. Our goal is useful both from the point of view of logical frameworks and of object-calculi under the following respects. First, the encoding in a logical framework forces to spell out in full detail all aspects of an object system. This enlightens problematic issues which are skipped on the paper, giving the possibility to identify and fix peculiar idiosyncrasies. Moreover, logical frameworks allow the user to (re)formulate the object system, taking full advantage of the proof-theoretical concepts provided by modern coinductive type theories. Thus, the logical frameworks perspective may suggest alternative, and cleaner, definitions of the same systems. In this scenario, a crucial aspect of our investigation is the style of logics for encoding the semantics of $\mathbf{imp}\varsigma$ in Coq.

A very successful approach for representing the operational semantics of programming languages is Plotkin's *Structural Operational Semantics* (SOS) [Plo81]. This approach reduces all computational elaboration and evaluation process to a formal logic derivation within a formal system, with the advantage that the specification is syntax-directed and easy to understand. However, even if this formalism overcomes many of the defects of other ones —as automata, or definitional interpreters— the explicit presence of the *environment* (binding identifiers to various entities, as values, or types) in the judgments can be seen as a limitation in the economy of proofs.

The SOS approach has been studied in depth by Kahn and his coworkers, so that the refined method of *Natural Semantics* (NS) [Kah87] has arisen. The general idea is to provide a unified manner for presenting several aspects of the semantics of programming languages, as static and dynamic semantics, interpretation, and so on. This kind of formalism supplies axioms and inference rules for characterizing the various semantic predicates. The approach is proof-theoretic, thus semantic descriptions can be seen in two ways: basic facts or descriptions of possible (non deterministic) computations.

Concluding, we feel that the formalism of *Natural Operational Semantics* (NOS) [BH90], a refinement of the primary natural semantics approach, is better suited w.r.t. the formalization in a logical framework. This technique allows to delegate the handling of structures which obey a stack discipline (such as the environments) to the metalanguage, by means of hypothetical-general premises à la Martin-Löf. Therefore, environments do not appear explicitly in judgments and proofs, which become appreciably simpler, anymore. The NOS approach derives assertions in natural semantics under assumptions, i.e. it adopts a Gentzen's natural deduction style of proof, where the hypothetical premises consist of assumptions about the values of the free variables.

Part II

Real Numbers

Chapter 3

A co-inductive model of the Real Numbers in Coq

We represent the Constructive Real Numbers using streams, i.e. infinite sequences, of digits. We adopt co-inductive types and co-recursive proofs, which permit to work naturally on streams in the proof assistant Coq.

3.1 Real numbers in logical frameworks

The construction of the real numbers in logical frameworks is a first step towards a rigorous approach to the reliability of computer programs used for high-precision numerical computations. Many applicative areas of computer science, physics, engineering, aeronautics and other disciplines would benefit of this. In fact, machines are largely used for computing on real numbers, as for example in numerical analysis, computational geometry, hybrid systems, computer algebra and many other applications requiring continuous datatypes and mathematical infrastructure.

Many classical constructions of the real numbers exist in literature: Cauchy sequences of rational numbers, Cauchy sequences of rational p-adic numbers, Dedekind cuts in the field of rationals, infinite decimal expansions, and so on. All these methods are equivalent, in the sense that give rise to isomorphic structures. Several of these constructions can be also formulated in a *constructive* approach to mathematics, but in this case we do not obtain always isomorphic structures —e.g. constructive reals via Cauchy sequences differ from constructive reals through Dedekind's cuts [TvD88].

We have already motivated in section 2.1 that we intend to adhere to the constructive approach to mathematics, i.e. —according to Brouwer [Bro07] and Bishop [Bis67]— algorithmic, or computational, mathematics, which is, following seminal ideas of Martin-Löf [NPS90], very close to computer science.

Since the real numbers are potentially "infinitely informative" datatypes, a natural way to represent them is the use of streams, viz infinite sequences, of digits. This choice is almost universally considered the more direct solution [Har96, Esc00], because it avoids to have to introduce the rational numbers (as in the case of Cantor's and Dedekind methods). Moreover, digit expansions are quite well-suited for implementing algorithms and doing computations.

The solution of using streams for representing real numbers is a typical choice adopted

in *exact computation* [PEE97, Sim98, Wei00]. This technique is one of the possible solutions introduced to overcome the unreliability of the more common floating-point approach: it avoids completely the round-off practice and permits to obtain correct results from computations without having to carry out any independent error analysis.

It turns out also that streams are a datatype immediately available in many programming languages, thus permitting a simple and natural implementation of the exact computation. It is typically carried out on machines via lazy functional programming languages: streams are real number generators, which are evaluated (lazily) from left to right, namely from the most meaningful digits. Since exact computation is motivated by software reliability reasons, it is important to certify the correctness of the algorithms working on the real numbers.

In order to approach streams and exact computation we need tools for defining and reasoning on infinite entities. A modern solution, partially supported by the current generation of logical frameworks, is to adopt co-inductive definition and proof principles, an approach allowing for a suitable treatment of infinite and circular objects [JR97, Len99]. In particular, the Calculus of Inductive and Co-inductive Constructions ($CC^{(Co)Ind}$) [PM93, Gim94] is a type theory providing the user with co-inductive principles and proof systems, thus allowing for the treatment of this kind of objects.

The $CC^{(Co)Ind}$ is mechanized in the system Coq [INR03], the only proof assistant which embarks a proof tactic (`Cofix`) specific for proving co-inductive assertions, that is, for building infinite proofs. The alternative, less direct and comfortable, is to choose suitable co-inductive set-theoretical principles and to encode them using the specification language: this can be done, in principle, in any sufficiently powerful logical framework. Coq can be used both as environment for giving specifications and developing machine-checked mathematical proofs and as a programming tool for the construction of certified programs. The standard logical setting of Coq is intuitionistic: this adheres to the constructive perspective, thus supplying the extraction of certified algorithms feature out of formal proofs about their behavior.

The goal. We pick out the proof assistant Coq for experimenting a co-inductive implementation of the constructive real numbers and for certifying exact algorithms on them. Our aim is to introduce a concrete data structure for the reals and to prove it is adequate using the standard constructive logical setting of the system.

The main line of our research is the following: we start representing real numbers by streams; then we define (co-)inductively on streams the notions of strict order (Less), addition (Add) and multiplication (Mult), and we implement addition and multiplication functions working on streams. In the next chapter we will prove formally in Coq that these definitions satisfy a standard set of properties and we will claim that such properties can be taken as axioms for the constructive real numbers.

3.2 Construction via streams

We build the real numbers using streams as *infinite positional expansions*, that is, infinite sequences of digits, and we work with constructive logic. An immediate difficulty is the well-known phenomenon that standard positional notations *are not computationally adequate* w.r.t. arithmetic operations [Bro24], see section 2.3.

An usual solution for this problem is to adopt *redundant* notations, where a real number enjoys more than one representation: typically, infinitely many ones. We decide to use here a *signed-digit* notation: we add the negative digit -1 to the binary digits 0 and 1 of the standard binary notation, maintaining 2 as the value for the base.

Data structures. We introduce the basic ingredients of the work, following the material presented in [CDG00], which is however considerably extended in the present document. In order to explain and motivate our definitions, we refer to a field of the real numbers $\mathbb{R}$: this use of a given field is not essential in our construction, but helps to understand the intended meaning of the data structures.

Definition 3.1 ***(Ternary streams)***
Let str be the set of the infinite sequences built of ternary digits:

$$str = \{a_1 : a_2 : a_3 : \ldots \mid \forall i \in \mathbb{N}^+.\ a_i \in \{0, 1, -1\}\}$$

The elements of str represent the real numbers via the interpretation function $[\![\ ]\!]_{str} : str \to \mathbb{R}$*, defined by:*

$$[\![a_1 : a_2 : a_3 : \ldots\ ]\!]_{str} = \sum_{i \in \mathbb{N}^+} a_i \cdot 2^{-i}$$

From now on we use $a, b, c, \ldots$ as metavariables ranging on ternary digits and $x, y, z, \ldots$ as metavariables for streams.

Using ternary streams, we can represent any real number belonging to the closed interval $[-1, 1]$. It is not difficult to see that any element of the open interval $(-1, 1)$ is represented by an infinite number of different streams: more precisely, each binary rational $m/2^n$ has countably infinite representations, and every other number has uncountable many representations. In order to dispose of arbitrarily large reals, it is necessary to use an exponent-mantissa notation: namely, we encode a real number by a pair $\langle natural, stream \rangle$, that we call *R-pair*.

Definition 3.2 ***(R-pairs)***
Let R be the set $(\mathbb{N} \times str)$*. The elements of R represent the real numbers via the interpretation function* $[\![\ ]\!]_R : R \to \mathbb{R}$*, defined by:*

$$[\![\langle n, x \rangle]\!]_R = 2^n \cdot [\![x]\!]_{str}$$

We will use $r, s, t, \ldots$ as metavariables ranging on R.

In order to complete our construction it is necessary to provide R with an *order* relation and a *field* structure: actually, the real line is completely determined by the binary *strict order* relation $(<)$ and the arithmetic operations of *addition* and *multiplication*.

We have considered several different possible characterizations for order, addition and multiplication along our research: at the end, we have chosen to describe not only the order, but also the operations using predicates and not functions. This choice is due to the fact that predicates are simpler to specify and work with. An intuitive motivation for this is that functions are requested to be "productive" —i.e. they must supply a method to effectively produce the result, given the input; on the contrary, a predicate just specifies the constrains that the output has to satisfy w.r.t. the input. Thus it is a simpler task to prove the formal properties of the predicates. Anyway, we will introduce the functions as

well and we will prove they are coherent with respect to the predicates. One can interpret this fact saying that the implementation (described by algorithms, i.e. functions) satisfies the specification (described by predicates).

We have devised that the length and the complexity of the formal proofs about the predicates are greatly affected by the pattern of their specifications: very often the proofs are carried out by structural (co-)induction on the derivations, thus the number of the cases to consider grows together with the number of constructors of the predicate involved. In order to simplify the proofs, we have formalized the (co-)inductive predicates using at most two constructors, hence reducing the cases to address.

We claim that these considerations about specifications and proofs have general meaning and are not specific to the particular proof assistant we have used: in fact, the proofs developed in Coq are just a completely detailed version of the proofs that we would write with paper and pencil.

Judgments. Let now resume our goal: to define order, addition and multiplication. The strict order relation ($<$) is defined by *induction*: this is possible because, given two R-pairs, we can semi-decide whether the first is smaller than the second just by examining a finite number of digits. The binary strict order relation on streams is defined in terms of an auxiliary ternary relation $less\_aux \subseteq (str \times str \times Z)$, whose intended meaning is:

$$less\_aux(x, y, i) \;\Leftrightarrow\; (\llbracket x \rrbracket_{str} < \llbracket y \rrbracket_{str} + i)$$

This auxiliary predicate permits to simplify the management of the order: the use of the integer parameter i allows to obtain simpler proofs, because the extensive case analysis on the ternary digits is replaced by automated proofs over integers. The main binary predicate on streams $less_{str} \subseteq (str \times str)$ is defined fixing the value of the integer parameter to 0:

$$less_{str}(x, y) \;\Leftrightarrow\; (\llbracket x \rrbracket_{str} < \llbracket y \rrbracket_{str})$$

Definition 3.3 ***(Order on streams)***
The predicate $less\_aux \subseteq (str \times str \times Z)$ *is defined by induction:*

$$\text{(less-base)}\ \frac{big \leq i}{less\_aux(x, y, i)} \qquad where\ big = 32$$

$$\text{(less-ind)}\ \frac{less\_aux(x,\ y,\ (2i + b - a))}{less\_aux(a : x,\ b : y,\ i)}$$

The predicate on streams $less_{str} \subseteq (str \times str)$ *is defined by:*

$$less_{str}(x, y) \;\triangleq\; less\_aux(x, y, 0)$$

This definition requires some additional explanations. It is easy to see, referring to the intended meaning, that $less\_aux(x, y, i)$ is valid for any value of the parameter i greater than 2: a natural choice for the constant *big* would be the integer 3, but it turns out that any greater value gives rise to an equivalent definition. We have found that greater values simplify several proofs built by structural induction on the judgment *less_aux*: in fact, the $(less{-}base)$ rule has a stronger premise as the value of *big* grows, and so it can be

proved in a simpler way. By our experience there is no natural choice for *big*; we have arbitrarily fixed it to 32. This value is sufficiently large to avoid the need of not strictly necessary extra arguments in the proofs we have developed.

It is immediate to see that the base rule is correct. The induction rule can be informally justified by means of a simple calculation:

$$\begin{array}{ll} [\![a:x]\!]_{str} \;<\; ([\![b:y]\!]_{str} + i) & \Leftrightarrow \\ a/2 + ([\![x]\!]_{str}/2) \;<\; b/2 + ([\![y]\!]_{str}/2) + i & \Leftrightarrow \\ [\![x]\!]_{str} \;<\; [\![y]\!]_{str} + 2i + b - a & \end{array}$$

The strict order relation can be easily extended to R-pairs through the predicate $Less \subseteq (R \times R)$, delegating the comparison to the stream component:

$$Less(r, s) \;\Leftrightarrow\; ([\![r]\!]_R < [\![s]\!]_R)$$

Definition 3.4 ***(Order on R-pairs)***
The predicate $Less \subseteq (R \times R)$ *is defined by:*

$$Less(\langle m, x\rangle, \langle n, y\rangle) \;\triangleq\; less_{str}(\underbrace{0:\ldots:0}_{n} : x,\; \underbrace{0:\ldots:0}_{m} : y)$$

Differently from the order, the arithmetic predicates are defined by *co-induction*, because the process of adding and multiplying two real numbers is not terminating. Co-inductive predicates give rise to assertions that have to be proved by an infinite application of the corresponding constructors [Coq93, Gim94].

The predicates of addition and multiplication share the following pattern:

$$predicate(operand_1,\; operand_2,\; result)$$

We start from addition: as done for the order relation, we define first an auxiliary predicate on streams. The relation $add\_aux \subseteq (str \times str \times str \times Z)$ has intended meaning:

$$add\_aux(x, y, z, i) \;\Leftrightarrow\; ([\![x]\!]_{str} + [\![y]\!]_{str}) = ([\![z]\!]_{str} + i)$$

Then we obtain the main ternary predicate on streams by fixing the value of the integer parameter to 0. And so we can give also the addition predicate for R-pairs.

Definition 3.5 ***(Addition)***
The predicate $add\_aux \subseteq (str \times str \times str \times Z)$ *is defined by co-induction:*

$$\text{(add-coind)}\;\; \frac{add\_aux(x,\; y,\; z,\; (2i + c - a - b)) \quad (-big < i < big)}{add\_aux(a:x,\; b:y,\; c:z,\; i)}$$

The addition predicate on streams $add_{str} \subseteq (str \times str \times str)$ *is defined by:*

$$add_{str}(x, y, z) \quad \triangleq \quad add\_aux(x, y, z, 0)$$

The addition predicate on R-pairs $Add \subseteq (R \times R \times R)$ *is defined by:*

$$Add(\langle m, x\rangle, \langle n, y\rangle, \langle p, z\rangle) \quad \triangleq \quad add_{str}(\underbrace{0\ldots0}_{n+p} : x,\; \underbrace{0:\ldots:0}_{m+p} : y,\; \underbrace{0:\ldots:0}_{m+n} : z)$$

The side-condition $(-big < i < big)$ has been introduced in order to make the relation add_aux not total —otherwise one can easily prove the assertion $add\_aux(x, y, z, i)$ for any 4-tuple x, y, z, i. Similarly to the order, values of big greater than 3 give rise to equivalent definitions, but lead to simpler proofs.

The co-inductive rule $(add{-}coind)$ can be informally justified by the calculation:

$$\begin{array}{lll} (\llbracket a : x \rrbracket_{str} + \llbracket b : y \rrbracket_{str}) \;=\; (\llbracket c : z \rrbracket_{str} + i) & & \Leftrightarrow \\ a/2 + (\llbracket x \rrbracket_{str}/2) + b/2 + (\llbracket y \rrbracket_{str}/2) \;=\; c/2 + (\llbracket z \rrbracket_{str}/2) + i & & \Leftrightarrow \\ \llbracket x \rrbracket_{str} + \llbracket y \rrbracket_{str} \;=\; \llbracket z \rrbracket_{str} + 2i + c - a - b & & \end{array}$$

As far as the multiplication is concerned, we introduce a preliminary multiplication *function* between signed digits and streams $times_{d,str} : \{0, -1, 1\} \times str \to str$, whose intended meaning is the following:

$$\llbracket times_{d,str}(a, x) \rrbracket_{str} \;=\; a \cdot \llbracket x \rrbracket_{str}$$

As usual, the multiplication operation can be reduced to the addition. So we can define directly a ternary multiplication predicate on streams $mult_{str} \subseteq (str \times str \times str)$ and R-pairs $Mult \subseteq (R \times R \times R)$, that have the natural intended meaning.

Definition 3.6 ***(Multiplication)***
The function $times_{d,str} : \{-1, 0, 1\} \times str \to str$ is defined by co-recursion:

$$times_{d,str}(a, (b : x)) \;\triangleq\; (a \cdot b) : (times_{d,str}(a, x))$$

The multiplication predicate on streams $mult_{str} \subseteq (str \times str \times str)$ is defined by co-induction:

$$\text{(mult-coind)}\ \frac{mult_{str}(x, y, w) \quad add_{str}(0 : times_{d,str}(a, y),\ 0 : w,\ z)}{mult_{str}(a : x,\ y,\ z)}$$

The multiplication predicate on R-pairs $Mult \subseteq (R \times R \times R)$ is defined by:

$$Mult(\langle m, x\rangle, \langle n, y\rangle, \langle p, z\rangle) \quad \triangleq \quad mult_{str}(\underbrace{0 : \ldots : 0}_{p} : x,\ y,\ \underbrace{0 : \ldots : 0}_{m+n} : z)$$

The co-inductive rule $(mult{-}coind)$ can be informally justified by calculations similar to those ones we have detailed for order and addition:

$$\begin{array}{lll} (\llbracket a : x \rrbracket_{str} \cdot \llbracket y \rrbracket_{str}) \;=\; \llbracket z \rrbracket_{str} & & \Leftrightarrow \\ (a/2 + \llbracket x \rrbracket_{str}/2) \cdot \llbracket y \rrbracket_{str} \;= \llbracket z \rrbracket_{str} & & \Leftrightarrow \\ (a \cdot \llbracket y \rrbracket_{str})/2 + (\llbracket x \rrbracket_{str} \cdot \llbracket y \rrbracket_{str})/2 \;= \llbracket z \rrbracket_{str} & & \end{array}$$

Research lines. Now we have introduced R-pairs for representing real numbers, we detail the next aims of our research:

- to devise the standard properties of the constructive real numbers;
- to prove the adequacy of our construction. That is, to prove that the full structure of R-pairs together with the order, addition and multiplication satisfies the standard properties of the constructive real numbers;
- to implement and certify exact real arithmetic algorithms.

The successful addressing of these points assures that the implementation given is (i.e. R-pairs are) a suitable model of the constructive real numbers; and, moreover, that we dispose of reliable algorithms working on the implementation.

3.3 Adequacy of the encoding

In this section we start addressing the question about the *adequacy* of the representation given in the previous section. We state two approaches for justifying our construction: the first argument is internal-axiomatic, the second one is external-semantic.

Following the first approach, we show that the notions of order, addition and multiplication are sufficient for capturing the standard properties of the constructive reals, and we prove that our implementation satisfies all these properties. The proof is carried out in Coq and will be presented in the next chapter.

According to the semantic approach, we justify the predicates of order, addition and multiplication by proving that their specification is sound and complete with respect to an external model of the reals $\mathbb{R}$. We present immediately this second argument.

External adequacy. In the construction of section 3.2 we have specified the intended meaning of the R-pairs and predicates referring to an external model of the real numbers $\mathbb{R}$. Therefore we can justify our encoding by proving that the specifications of order, addition and multiplication are sound and complete with respect to $\mathbb{R}$. For carrying out this proof we make use of simple arithmetic properties valid both for the constructive and the classical real numbers: so doing, we do not need to specify whether the external model $\mathbb{R}$ is classical or constructive [TvD88].

This proof has not been formalized in Coq, but we conjecture that this could be possible in the case one disposes of a suitable formalization of the real numbers —for example the classical axiomatization provided by the library `Reals` [May01].

Proposition 3.1 ***(Adequacy of order)***
The rules for the order predicate are sound and complete.
That is, given two R-pairs r, s: $Less(r, s)$ can be derived if and only if $([\![r]\!]_R < [\![s]\!]_R)$.

Proof. It suffices to prove the soundness and completeness of the rules for the auxiliary predicate *less_aux*, defined on streams: the proposition about R-pairs follows easily from it. We show that $\forall x, y \in str, \forall i \in Z.\ less\_aux(x, y, i)$ is derived if and only if $([\![x]\!]_{str} < [\![y]\!]_{str} + i)$. Notice that the proof works for any value of the integer constant *big* greater than 2.

($\Rightarrow$). It is quite simple to prove that the two rules ($less{-}base$) and ($less{-}ind$) are sound:

$$\text{(less-base)}\ \frac{big \leq i}{less\_aux(x, y, i)} \qquad \text{(less-ind)}\ \frac{less\_aux(x,\ y,\ (2i + b - a))}{less\_aux(a : x,\ b : y,\ i)}$$

The base rule is sound because every stream x represents a real number belonging to the interval $[-1, 1]$: thus $[\![x]\!]_{str} < [\![y]\!]_{str} + 3 \leq [\![y]\!]_{str} + big \leq [\![y]\!]_{str} + i$.

The induction rule is sound because $[\![a : x]\!]_{str} = a/2 + [\![x]\!]_{str}/2 < b/2 + [\![y]\!]_{str}/2 + i \Leftrightarrow [\![x]\!]_{str} < [\![y]\!]_{str} + (2i + b - a)$, which is the inductive hypothesis.

($\Leftarrow$). The proof of the completeness is less obvious. We need to choose a natural number k such that $(big{+}6 < 2^k)$ and to prove, by induction on n, that if $([\![x]\!]_{str}{+}2^{k-n} < [\![y]\!]_{str}{+}i)$ holds, then the assertion $less\_aux(x, y, i)$ can be derived. The completeness property follows immediately from this statement, because if $([\![x]\!]_{str} < [\![y]\!]_{str} + i)$, then there exists

a natural number n such that $([\![x]\!]_{str} + 2^{-n} < [\![y]\!]_{str} + i)$. Let us assume $x \equiv (a : x_0)$, $y \equiv (b : y_0)$.

Base step ($n = 0$). By hypothesis $([\![x]\!]_{str} + 2^k < [\![y]\!]_{str} + i)$, and so $big < big + 4 < 2^k - 2 \leq 2^k + [\![x]\!]_{str} - [\![y]\!]_{str} < i$: therefore we derive $less\_aux(x, y, i)$ by the rule $(less{-}base)$. Concerning the induction rule, we have $a/2 + [\![x_0]\!]_{str}/2 + 2^k < b/2 + [\![y]\!]_{str}/2 + i \Leftrightarrow [\![x_0]\!]_{str} + 2^{k+1} < [\![y_0]\!]_{str} + (2i + b - a)$, that means $[\![x_0]\!]_{str} + 2^k < [\![y_0]\!]_{str} + (2i + b - a)$. We can conclude by induction hypothesis and using the rule $(less{-}ind)$.

Induction step ($n = m + 1$). We provide a derivation of $less\_aux(a : x_0,\ b : y_0,\ i)$, having by hypothesis $([\![a : x_0]\!]_{str} + 2^{k-(m+1)}) < ([\![b : y_0]\!]_{str} + i)$, that is $([\![x_0]\!]_{str} + 2^{k-m}) < ([\![y_0]\!]_{str} + (2i + b - a))$. We conclude by induction and using the $(less{-}ind)$ rule. □

We consider now the arithmetic predicates: it is interesting to notice that the co-inductive proof technique we adopt below is dual with respect to the inductive one used for the order.

Proposition 3.2 ***(Adequacy of addition)***
The rules for the addition predicate are sound and complete.
That is, given a triple of R-pairs r, s, t: $Add(r, s, t)$ can be derived by an infinite derivation if and only if $([\![r]\!]_R + [\![s]\!]_R = [\![t]\!]_R)$.

Proof. Like in the case of the order relation, it is sufficient to prove the soundness and completeness of the co-inductive rule for the auxiliary predicate add_aux:

$$\text{(add-coind)}\ \frac{add\_aux(x,\ y,\ z,\ (2i + c - a - b)) \quad (-big < i < big)}{add\_aux(a : x,\ b : y,\ c : z,\ i)}$$

We show that $\forall x, y, z \in str, \forall i \in Z.\ add\_aux(x, y, z, i)$ is derived by and infinite derivation if and only if $([\![x]\!]_{str} + [\![y]\!]_{str} = [\![z]\!]_{str} + i)$. Notice that the proof below works for any value of the constant big greater than 3.

($\Leftarrow$). It is quite simple to prove by co-induction that the rule $(add{-}coind)$ is complete. We prove that, under the hypothesis $([\![x]\!]_{str} + [\![y]\!]_{str} = [\![z]\!]_{str} + i)$, there exists a rule permitting to derive $add\_aux(x, y, z, i)$ using the judgment $add\_aux(x', y', z', i')$, whose arguments satisfy $([\![x']\!]_{str} + [\![y']\!]_{str} = [\![z']\!]_{str} + i')$. In this case the conclusion $add\_aux(x, y, z, i)$ can be obtained by an infinite proof using the co-inductive hypothesis.

Let be $x \equiv (a : x_0)$, $y \equiv (b : y_0)$, $z \equiv (c : z_0)$ and:

$$[\![a : x_0]\!]_{str} + [\![b : y_0]\!]_{str} = [\![c : z_0]\!]_{str} + i \tag{3.1}$$

By a simple calculation, we derive immediately both $[\![x_0]\!]_{str} + [\![y_0]\!]_{str} = [\![z_0]\!]_{str} + (2i + c - a - b)$ and $(-3 \leq i \leq 3)$. Then we use the rule $(add{-}coind)$ for deducing $add\_aux(a : x_0,\ b : y_0,\ c : z_0,\ i)$ from $add\_aux(x_0,\ y_0,\ z_0,\ 2i + c - a - b)$, and so we can conclude carrying out an infinite proof using the equation 3.1.

($\Rightarrow$). The proof of the soundness is a little more subtle. We cannot simply prove that the rule $(add{-}coind)$ is sound, because by using infinitely a rule which deduces valid conclusions from valid premises it is still possible to derive judgments that are not valid —for instance consider a rule for equality saying that from $(x = y)$ follows $(y = x)$, or the rule $(add{-}coind)$ itself, where the premise $(-big < i < big)$ has been removed.

Thus we need to use other arguments. First we choose a natural number k such that $(big + 3 \leq 2^k)$, then we prove by induction on n that if $add\_aux(x, y, z, i)$ can be derived, then:

$$|([\![x]\!]_{str} + [\![y]\!]_{str}) - ([\![z]\!]_{str} + i)| < 2^{k-n}$$

The soundness follows from this inequality, because two arbitrarily close real numbers must be equal. The inductive proof works as follows: let be $x \equiv (a : x_0)$, $y \equiv (b : y_0)$, $z \equiv (c : z_0)$; if the predicate $add\_aux(x, y, z, i)$ can be derived, it should be via the co-inductive rule $(add{-}coind)$. Therefore both the constraint $(-big < i < big)$ and the judgment $add\_aux(x_0,\ y_0,\ z_0,\ 2i + c - a - b)$ can be derived.

Base step $(n = 0)$. The thesis is immediate.

Induction step $(n = m + 1)$. We have by inductive hypothesis $|([\![x_0]\!]_{str} + [\![y_0]\!]_{str}) - ([\![z_0]\!]_{str} + 2i + c - a - b)| < 2^{k-m}$, thus we conclude the proof via a simple arithmetic calculation. □

Proposition 3.3 ***(Adequacy of multiplication)***
The rules for the multiplication predicate are sound and complete.
That is, given a triple of R-pairs r, s, t: $Mult(r, s, t)$ can be derived by an infinite derivation if and only if $([\![r]\!]_R \cdot [\![s]\!]_R = [\![t]\!]_R)$.

Proof. The proof follows from the adequacy of addition and works similarly to that one. □

Internal adequacy. We intend to use the proof assistant Coq for justifying our construction of the constructive real numbers. Hence we have to prove formally that our encoding through R-pairs, order, addition and multiplication satisfies all the standard properties of the constructive reals.

In order to carry out this proof, we have first to formalize in Coq the notions presented in section 3.2 and we should also dispose of the properties that characterize the constructive reals, i.e. a set of axioms. At the origin of our research we looked in the literature for a constructive axiomatization, but we did not find any standard one. So we decided to synthesize a set of axioms by ourselves, with the aim to be minimal, namely, of assuming the least number of primitive notions and properties. This minimality is interesting and useful both for theoretical reasons —the mathematical curiosity about an essential characterization of the constructive reals is addressed— and practical ones— a simpler test for checking models is provided. We will introduce, discuss and justify our axiomatization in the next chapter, when we will use it for addressing the internal adequacy of our construction of the reals.

3.4 Formalization in Coq

This section is devoted to the formalization in the proof assistant Coq of the basic structures and predicates we have introduced so far. In parallel, we present and discuss definition principles and proof techniques we use largely in the internal formal development of the next chapter.

Datatypes. The first step is to introduce concrete structures —namely types whose type is `Set`— for representing signed-digits, streams and R-pairs. The formalization is as follows:

```
Inductive digit : Set := mino : digit | zero : digit | one : digit.
CoInductive str : Set := cons : digit -> str -> str.
Inductive R : Set := pair : nat -> str -> R.
```

We define the three recursive types *digit*, *str* and R. Recursive types are characterized by constructors, or introduction rules, which specify the way of building their inhabitants. Terms in inductive types are built by a *finite* application of the introduction rules, whereas terms in co-inductive types can also be obtained by an *infinite* use of the rules: that is, inhabitants of co-inductive types are *potentially infinite* objects. In our case, we do only with infinite streams.

The system Coq automatically associates to inductive sets suitable definition and proof principles: it provides the user with the possibility of defining functions by primitive recursion on inductive terms and building proofs by structural induction on them.

The co-inductive case is not perfectly dual: Coq allows to build terms by co-recursion, but does not supply proof principles associated to co-inductive sets. Indeed, the system mechanizes the *guarded induction principle* of Coquand and Giménez [Coq93, Gim94], which is associated to co-inductive *predicates*, i.e. types whose signature ends with *Prop*. This is a proof schema for carrying out infinitely regressive proofs: formally speaking, it permits to use the conclusion of logical rules as an auxiliary hypothesis, provided it is applied within introduction rules. These rules guard the application of the co-inductive hypothesis, thus generating a step-by-step infinite logical process. This proof principle is implemented by the tactic `Cofix`. Notice that this explanation depends on the top-down proof practice of Coq, where the main goal is transformed into simpler goals using tactics, and a proof succeeds when all the current sub-goals are instances of axioms. The analogy between co-recursive objects and proofs relies on the propositions-as-types principle [How80, dB80]: proofs of co-inductive assertions are actually represented by co-recursive lazy (proof) terms.

Coming back to our formalization, signed-digits are encoded by an inductive type with three 0-ary constructors: this choice allows to define functions on digits by primitive recursion and to develop proofs by induction and case analysis.

Ternary streams are modeled through infinite sequences built of signed-digits: this encoding enables to define, by co-recursion, functions generating streams and having arbitrary domain. A trivial example is the construction of the infinite sequence formed using only the digit 0, obtained by the following definition:

```
CoFixpoint zeros : str := (cons zero zeros)
```

Notice that co-recursive functions as *zeros* can be seen as maximal fixed-point equations, or infinite repetition generation laws. A co-recursive equation is accepted by Coq if and only if the recursive call is *guarded (only) by constructors* [Coq93, Gim94, Gim98], in the sense that only constructors can participate to the generation of co-recursive terms. The guardedness condition guarantees the *strong normalization* property of the logical framework, which is crucial for the decidability of the *type-checking* and the *proof checking*:

infinite objects like *zeros* are constructed *lazily* in order to preserve the strong normalization; that is, they are expanded, according to their definition, only when arguments of a case analysis operation.

Finally, R-pairs are formalized by an inductive type, whose unique constructor collects in a single structure $\langle natural, stream \rangle$ the two primitive components. An alternative would be to model R-pairs by product types.

Predicates. The specification of order, addition and multiplication require two preliminary definitions. The function *code* maps the constructors of digits into corresponding built-in integer values; the function *appendO* "normalizes" streams (i.e. divides them by powers of two) by means of finite prefixes of 0 digits:

```
Definition code : digit -> Z := [a:digit]
          Cases a of mino => `-1` |  zero => `0` |  one  => `1`
          end.
Fixpoint appendO [n:nat] : str -> str := [x:str]
         Cases n of (O)   => x
                 |  (S m) => (cons zero (appendO m x))
         end.
```

The function *code* is defined by case analysis on the type of digits through the construct `Cases...of...end`, typical of functional programming languages. This mapping will allow us to automate integer calculations in the proof assistant. The function *appendO* is defined by recursion on natural numbers, whose constructors are `O: nat` and `S: nat -> nat`. Recursive functions are dual of co-recursive ones and have to satisfy the *guardedness by destructors* condition, requiring that recursive calls have to be performed on structurally smaller arguments. This property guarantees the termination of the algorithm and preserves the strong normalizability of terms in the logical framework.

We can give now the encoding of the main predicates *Less*, *Add* and *Mult*. Concerning the inductive order, we have the following hierarchy of declarations:

```
Inductive less_aux : str -> str -> Z -> Prop :=
          less_base: (x,y:tstr) (i:Z)
                     (`big <= i`) ->
                     (less_aux x y i)
    |     less_ind : (x,y:tstr) (a,b:treat) (i:Z)
                     (less_aux x y `2*i + (code b) - (code a)`) ->
                     (less_aux (cons a x) (cons b y) i).

Definition less_str : str -> str -> Prop := [x,y:str]
           (less_aux x y `0`).

Definition Less : R -> R -> Prop := [r,s:R]
           Cases r of (pair m x) => Cases s of (pair n y) =>
                      (less_str (appendO n x) (appendO m y))
           end end.
```

The inductive predicate *less_aux* provides the user with a new proof principle: it is actually possible to state assertions as $less\_aux(x, y, i)$ and to develop proofs by structural induction on the structure of their derivation. This technique will be largely used during the proof of the internal adequacy, which is addressed in the next chapter. The level of streams and R-pairs does not require any specific explanation.

As far as the co-inductive addition is concerned, the hierarchy of definitions is analogous; the only difference is that this time we specify a co-inductive relation:

```
CoInductive add_aux : str -> str -> str -> Z -> Prop :=
      add_coind : (x,y,z:str) (a,b,c:treat) (i:Z)
                  (add_aux x y z `2*i-(code a)-(code b)+(code c)`) ->
                  (`-big < i`) -> (`i < big`) ->
                  (add_aux (cons a x) (cons b y) (cons c z) i).

Definition add_str : str -> str -> str -> Prop := [x,y,z:str]
           (add_aux x y z `0`).

Definition add : real -> real -> real -> Prop := [r,s,p:real]
           Cases r of (pair m x) => Cases s of (pair n y) =>
           Cases p of (pair l z) => (add_str (appendO (plus n l) x)
                                             (appendO (plus m l) y)
                                             (appendO (plus m n) z))
           end end end.
```

As far as the co-inductive multiplication is concerned, we want just to remark that we implement the auxiliary function $times_{d,str}$ using a co-recursive function. Notice that the recursive call is guarded by constructors:

```
Definition dual : digit -> digit -> digit := [a,b:digit]
           Cases a of zero => zero
                   |  mino => one
                   |  one  => mino end.

Definition times_d : digit -> digit -> digit := [a,b:digit]
           Cases a of zero => zero
                   |  mino => (dual b)
                   |  one  => b end.

CoFixpoint times_d_str : digit -> str -> str := [a:digit; x:str]
           Cases x of (cons b y) => (cons (times_d a b)
                                          (times_d_str a y)) end.

CoInductive mult_str  : str -> str -> str -> Prop :=
            mult_coind: (x,y,z,w:str) (a:digit)
                        (mult_str x y w) ->
                        (add_str (cons zero (times_d_str a y))
                                 (cons zero w) z) ->
```

```
                        (mult_str (cons a x) y z).

Definition mult : R -> R -> R -> Prop := [r,s,p:real]
           Cases r of (pair m x) => Cases s of (pair n y) =>
           Cases p of (pair l z) => (add_str (appendO l x) (y)
                                             (appendO (plus m n) z))
           end end end.
```

Example. We detail now a full example about a co-inductive development in Coq, using streams, co-recursive functions and a co-inductive predicate.

First we introduce the co-recursive functions *odd* and *even* that, taken a stream as input, return respectively streams built by the elements in odd and even positions:

```
CoFixpoint odd : str -> str := [x:str]
           Cases x of (cons a y) => Cases y of (cons b z) =>
           (cons a (odd z)) end end.
CoFixpoint even : str -> str := [x:str]
           Cases x of (cons a y) => Cases y of (cons b z) =>
           (cons b (even z)) end end.
```

Next we define a co-recursive function *merge* that, given two streams, renders the stream built taking elements alternatively in its arguments:

```
CoFixpoint merge : str -> str -> str := [x,y:str]
           Cases x of (cons a z) => (cons a (merge y z)) end.
```

Then we model the co-inductive point-wise equality on streams; that is, an extensional equality syntactical check, captured with Leibniz's equality:

```
CoInductive Eq_str : str -> str -> Prop :=
            eq_c   : (x,y:str) (a:digit)
                     (Eq_str x y) -> (Eq_str (cons a x) (cons a y)).
```

Thus we are ready to state and prove the co-inductive assertion telling that every stream is point-wise equal to its transformation through the combination of the functions *odd*, *even* and *merge*:

```
(x:str) (Eq_str (merge (odd x) (even x)) (x))
```

This statement has to be proved invoking the proof editor of Coq and using the `Cofix` tactic. First we assume the thesis as hypothesis:

```
 co_hp : (x:str)(Eq_str (merge (odd x) (even x)) x)
 ============================
 (x:str)(Eq_str (merge (odd x) (even x)) x)
```

Then we perform twice a case analysis on the argument stream x:

```
co_hp : (x:str)(Eq_str (merge (odd x) (even x)) x)
a : digit
b : digit
```

```
z : str
============================
(Eq_str (merge (odd (cons a (cons b z)))
               (even (cons a (cons b z))))
        (cons a (cons b z)))
```

We can now compute by the three co-recursive functions:

```
co_hp : (x:str)(Eq_str (merge (odd x) (even x)) x)
a : digit
b : digit
z : str
============================
(Eq_str (cons a (cons b (merge (odd z) (even z))))
        (cons a (cons b z)))
```

Next we can apply twice the constructor *eq_c*, thus consuming input and obtaining:

```
co_hp : (x:str)(Eq_str (merge (odd x) (even x)) x)
z : str
============================
(Eq_str (merge (odd z) (even z)) z)
```

Finally we are allowed to apply the co-inductive hypothesis, because the use of the constructor *eq_c* has provided an initial step of the proof. So the recursive call consists in the infinitely regressive repetition of the first step, thus permitting to conclude. The whole formal development can be summarized as follows:

$$\begin{array}{lcl}
\texttt{merge}(\texttt{odd}(a:b:z),\texttt{even}(a:b:z)) & \cong & (a:b:z)\\
\texttt{merge}(a:\texttt{odd}(z),b:\texttt{even}(z)) & \cong & (a:b:z)\\
(a:\texttt{merge}(b:\texttt{even}(z),\texttt{odd}(z))) & \cong & (a:b:z)\\
(a:b:\texttt{merge}(\texttt{odd}(z),\texttt{even}(z))) & \cong & (a:b:z)\\
(b:\texttt{merge}(\texttt{odd}(z),\texttt{even}(z))) & \cong & (b:z)\\
\texttt{merge}(\texttt{odd}(z),\texttt{even}(z)) & \cong & (z)\\
 & Q.E.D. &
\end{array}$$

This proof is very short, but in general it may be difficult to ascertain if we are applying the co-inductive hypothesis in the right way; thus Coq provides the command `Guarded`, which automatically checks the condition on behalf of the user, notifying the result of the operation.

The usefulness of the `Cofix` tactic must not be underestimated, since it allows building proofs of co-inductive assertions in a very natural way. In fact, especially in the case of complex co-inductive predicates (e.g. bisimulations of process algebras), it is usually much more complicated to imagine a priori the needed bisimulation than building it progressively during a top down proof, as witnessed in [HMS01a].

3.5 Redundancy and equivalence

In our setting the real numbers are represented with *redundancy*: in fact an infinite choice of R-pairs denoting the same number is available. Thus it is natural to define an *equivalence*

relation on R-pairs. It turns out that in constructive analysis it is possible to describe the equivalence relation on real numbers by means of the strict order relation.

Definition 3.7 ***(Inductive equivalence)***
The inductive equivalence predicate $Equal_{ind} \subseteq (R \times R)$ *is defined by:*

$$Equal_{ind}(r,s) \quad \triangleq \quad \neg Less(r,s) \wedge \neg Less(s,r)$$

This definition enforces the choice of using the strict order as basic notion for constructive real numbers.

It is interesting to notice that the equivalence relation can also be defined directly via a co-inductive predicate. Following such an approach, it is convenient, as usual, to introduce first an auxiliary predicate $equal\_aux \subseteq (str \times str \times Z)$, which has intended meaning:

$$equal\_aux(x,y,i) \;\Leftrightarrow\; (\llbracket x \rrbracket_{str} = \llbracket y \rrbracket_{str} + i)$$

Definition 3.8 ***(Co-inductive equivalence)***
The predicate $equal\_aux \subseteq (str \times str \times Z)$ *is defined by co-induction:*

$$\text{(equal-coind)}\ \frac{equal\_aux(x,\ y,\ (2i+b-a)) \quad (-big < i < big)}{equal\_aux(a:x,\ b:y,\ i)}$$

The equivalence predicate on streams $equal_{str} \subseteq (str \times str)$ *is defined by:*

$$equal_{str}(x,y) \;\triangleq\; equal\_aux(x,y,0)$$

The equivalence predicate on R-pairs $Equal_{coind} \subseteq (R \times R)$ *is defined by:*

$$Equal_{coind}(\langle m,x\rangle,\langle n,y\rangle) \;\triangleq\; equal_{str}(\underbrace{0:\ldots:0}_{n}:x,\ \underbrace{0:\ldots:0}_{m}:y)$$

The above definitions are very similar to those ones given for the addition predicate: in fact, if we fix the first argument of the auxiliary addition predicate to a stream of $0s$, we can see the equivalence as a particular case of addition. Therefore the predicate $equal\_aux$ can be justified by the same arguments used for add_aux.

The crucial property concerning the equivalence is that the inductive and the co-inductive definitions are equivalent. In order to prove this result, it is convenient to introduce the *less or equal* relation ($\leq$). Such a notion is modeled by co-induction via the auxiliary predicate $leq\_aux \subseteq (str \times str \times Z)$:

$$leq\_aux(x,y,i) \;\Leftrightarrow\; (\llbracket x \rrbracket_{str} \leq \llbracket y \rrbracket_{str} + i)$$

Definition 3.9 ***(Co-inductive weak order)***
The predicate $leq\_aux \subseteq (str \times str \times Z)$ *is defined by co-induction:*

$$\text{(leq-coind)}\ \frac{leq\_aux(x,\ y,\ (2i+b-a)) \quad (-big < i)}{leq\_aux(a:x,\ b:y,\ i)}$$

The predicate on streams $leq_{str} \subseteq (str \times str)$ *is defined by:*

$$leq_{str}(x,y) \;\triangleq\; leq\_aux(x,y,0)$$

The predicate on R-pairs $Leq_{coind} \subseteq (R \times R)$ *is defined by:*

$$Leq_{coind}(\langle m,x\rangle,\langle n,y\rangle) \;\triangleq\; leq_{str}(\underbrace{0:\ldots:0}_{n}:x,\ \underbrace{0:\ldots:0}_{m}:y)$$

The following preliminary lemma collects some elementary properties connecting the two order relations with the co-inductive equivalence. The proofs are carried out by standard techniques, that we adopt from now on for proving inductive and co-inductive assertions.

Lemma 3.1 ***(Orders and co-inductive equivalence)***
Let be r and s R-pairs. Then:

(i). $\neg Less(r,s) \Rightarrow Leq(s,r)$

(ii). $Leq(r,s) \Rightarrow \neg Less(s,r)$

(iii). $Equal_{coind}(r,s) \Rightarrow Leq(r,s) \wedge Leq(s,r)$

(iv). $Leq(r,s) \wedge Leq(s,r) \Rightarrow Equal_{coind}(r,s)$

Proof. The lemma is proved formally in the Coq system. All the points are first managed at stream level, then the proof is lifted to the R-pair level.

(i). We prove by co-induction: $\forall x, y \in str,\ \forall i \in Z.\ \neg less\_aux(x,y,-i) \Rightarrow leq\_aux(y,x,i)$. That is, $leq\_aux(b:w, a:z, i)$ follows from $less\_aux(a:z, b:w, -i)$. The former statement can be derived from $leq\_aux(w,z,2i+a-b)$ and $(-big < i)$ using the rule $(leq{-}coind)$.

For proving the first subgoal first we apply the co-inductive hypothesis, thus reducing to $\neg less\_aux(z,w,-2i-a+b)$; that is, we have to prove $less\_aux(z,w,-2i-a+b) \Rightarrow less\_aux(a:z, b:w, -i)$. We conclude using the constructor $(less-ind)$.

We solve the second subgoal $\neg less\_aux(a:z, b:w, -i) \Rightarrow (-big < i)$ through the auxiliary lemma $(2 < i) \Rightarrow less\_aux(x,y,i)$, which, in turn, can be proved by case analysis.

Therefore, fixing $i = 0$, we have $\neg less_{str}(r,s) \Rightarrow leq_{str}(s,r)$. And finally, using this result and exploiting the definitions of $Less$ and Leq, we obtain $\neg Less(r,s) \Rightarrow Leq(s,r)$.

(ii). The proof technique is dual with respect to point (i): this time we prove preliminary by induction that $\forall x, y \in str,\ \forall i \in Z.\ less\_aux(y,x,i) \Rightarrow \neg leq\_aux(x,y,-i)$, which is the same of $\forall x, y \in str,\ \forall i \in Z.\ leq\_aux(x,y,-i) \Rightarrow \neg less\_aux(y,x,i)$. That is, $less\_aux(y,x,i)$ and $leq\_aux(x,y,-i)$ are in contradiction. Reasoning by induction on the derivation of $less\_aux(y,x,i)$, where $y \equiv (b:w)$, $x \equiv (a:z)$, we are presented with two cases.

If the last constructor used is $(less{-}base)$, it is $(big \leq i)$; but $leq\_aux(x,y,-i)$ has to be derived by the (only) constructor $(leq{-}coind)$, namely from $(-big < -i)$, so we obtain a contradiction.

Concerning the $(less{-}ind)$ constructor, we have by induction $\neg leq\_aux(z,w,-2i-b+a)$, and we have to infer $\neg leq\_aux(a:z, b:w, -i)$, namely $leq\_aux(a:z, b:w, -i) \Rightarrow leq\_aux(z,w,-2i-b+a)$. We can conclude observing that the premise has to be derived by the rule $(leq{-}coind)$, i.e. from an hypothesis coinciding with the goal.

At this point we can gain the R-pair level as in the case (i).

(iii). We prove separately by co-induction $equal\_aux(x,y,i) \Rightarrow leq\_aux(x,y,i)$ and $equal\_aux(x,y,i) \Rightarrow leq\_aux(y,x,-i)$. The proofs are straightforward.

Concerning the first one, we observe that the premise $equal\_aux(a:z, b:w, i)$ has to be derived from $equal\_aux(z,w,2i+b-a)$ and $(-big < i < big)$. So we can apply the rule $(leq{-}coind)$, obtaining $leq\_aux(z,w,2i+b-a)$, and then we conclude by co-inductive hypothesis. The proof of the implication $equal\_aux(x,y,i) \Rightarrow leq\_aux(y,x,-i)$ is analogous.

Fixing $i = 0$ we have immediately $equal_{str}(x, y) \Rightarrow leq_{str}(x, y) \wedge leq_{str}(y, x)$, that means $Equal_{coind}(r, s) \Rightarrow Leq(r, s) \wedge Leq(s, r)$.

(iv). The proof is completely similar to the one of point (iii). First we prove by co-induction that $leq\_aux(x, y, i) \wedge leq\_aux(y, x, -i) \Rightarrow equal\_aux(x, y, i)$, then we gain the R-pair level as usual. □

Therefore it is possible to see the weak order as a derived notion; in particular we can establish $Leq(r, s) \triangleq \neg Less(s, r)$. Now we can state and prove the main result of this section.

Theorem 3.1 ***(Inductive and co-inductive equivalence)***
Let r and s be R-pairs. Then:

$$Equal_{ind}(r, s) \;\Leftrightarrow\; Equal_{coind}(r, s)$$

Proof. ($\Rightarrow$) We have to prove that $\neg Less(r, s) \wedge \neg Less(s, r) \Rightarrow Equal_{coind}(r, s)$: by point (i) of previous lemma 3.1 we have both $Leq(r, s)$ and $Leq(s, r)$; we conclude through (iv).

($\Leftarrow$) We deduce both $Leq(r, s)$ and $Leq(s, r)$ by lemma 3.1.(iii), so the thesis arises via a double application of (ii). □

We claim that the correspondence between a co-inductive predicate and the negation of an inductive one is just an instance of a more general phenomenon. We conjecture that a large class of co-inductive predicates (but not all [Coq93]) —those defined through decidable side-conditions in the rules— are equivalent to negations of certain inductive ones.

3.6 Certification of exact algorithms

In section 3.2 we have motivated our preference for the description of the arithmetic operations by the use of predicates rather than by functions. Anyway, the predicates are used only in formal proofs, whereas functions play an essential role in the concrete implementation of the reals. The goal of this section is to relate formally each other the algorithms, given through functions, and the specifications, described by predicates.

We introduce exact algorithms for addition and multiplication that work on our implementation of the reals. Then we certify these algorithms, that is, we prove the correctness of the functions implementing them with respect to the predicates Add and $Mult$.

Addition. The addition of streams is defined via an auxiliary function $+_{aux} : (str \times str \times Z) \rightarrow str$, whose intended meaning is:

$$[\![+_{aux}(x, y, i)]\!]_{str} = \begin{cases} ([\![x]\!]_{str} + [\![y]\!]_{str} + i)/4 & \text{if } ([\![x]\!]_{str} + [\![y]\!]_{str} + i) \in [-4, 4] \\ (-1) & \text{if } ([\![x]\!]_{str} + [\![y]\!]_{str} + i) < -4 \\ (+1) & \text{if } ([\![x]\!]_{str} + [\![y]\!]_{str} + i) > 4 \end{cases}$$

We can now design the algorithms working directly on streams and R-pairs; we introduce two functions $+_{str} : (str \times str) \rightarrow str$ and $+_R : (R \times R) \rightarrow R$:

$$\begin{array}{lcl} [\![+_{str}(x, y)]\!]_{str} & = & ([\![x]\!]_{str} + [\![y]\!]_{str})/2 \\ [\![+_R(r, s)]\!]_R & = & ([\![r]\!]_R + [\![s]\!]_R) \end{array}$$

Notice that, since a single stream can represent only the real numbers in the interval $[-1, 1]$, the result of the addition between streams has to be normalized (divided) by a factor 2.

We give now the specification of all the hierarchy of addition functions; we remark that our algorithm has linear complexity: it just examines the first digit of the (stream) arguments.

Definition 3.10 ***(Addition function)***
The function $+_{aux} : (str \times str \times Z) \to str$ *is defined by co-recursion:*

$$\begin{array}{rcl} +_{aux}(a : x_0,\ b : y_0,\ i) & \triangleq & \textbf{\textit{let}}\ j := (2i + a + b)\ \textbf{\textit{in}} \\ & & \textbf{\textit{Cases}}\ j\ \textbf{\textit{of}} \\ & & \quad j \geq 2\ \Rightarrow\ (1 : +_{aux}(x_0,\ y_0,\ j - 4)) \\ & & \quad j \in [-1, 1]\ \Rightarrow\ (0 : +_{aux}(x_0,\ y_0,\ j)) \\ & & \quad j \leq -2\ \Rightarrow\ (-1 : +_{aux}(x_0,\ y_0,\ j + 4)) \\ & & \textbf{\textit{end}} \end{array}$$

The addition function on streams $+_{str} : (str \times str) \to str$ *is defined by:*

$$+_{str}(a : x_0,\ b : y_0) \triangleq +_{aux}(x_0,\ y_0,\ a + b)$$

The addition function on R-pairs $+_R : (R \times R) \to R$ *is defined by:*

$$+_R(\langle m, x\rangle, \langle n, y\rangle) \triangleq \langle m + n + 1,\ +_{str}(\underbrace{0 : \ldots : 0}_{n} : x,\ \underbrace{0 : \ldots : 0}_{m} : y)\rangle$$

Multiplication. The multiplication algorithm is defined in terms of the addition one. Also for multiplication it is convenient to use an auxiliary function $\times_{aux} : (str \times str \times str \times [-2, 2]) \to str$, with intended meaning:

$$[\![\times_{aux}(x, y, z, i)]\!]_{str} = (([\![x]\!]_{str} \cdot [\![y]\!]_{str}) + [\![z]\!]_{str} + i)/4$$

Next we need two extra functions $norm : (Z \times str) \to str$ and $times_4 : str \to str$:

$$[\![norm(i, x)]\!]_{str} = \begin{cases} [\![x]\!]_{str} + i & \text{if } ([\![x]\!]_{str} + i) \in [-1, 1] \\ (-1) & \text{if } ([\![x]\!]_{str} + i) < -1 \\ (+1) & \text{if } ([\![x]\!]_{str} + i) > 1 \end{cases}$$

$$[\![times_4(x)]\!]_{str} = \begin{cases} [\![x]\!]_{str} \cdot 4 & \text{if } ([\![x]\!]_{str} \cdot 4) \in [-1, 1] \\ (-1) & \text{if } ([\![x]\!]_{str} \cdot 4) < -1 \\ (+1) & \text{if } ([\![x]\!]_{str} \cdot 4) > 1 \end{cases}$$

We gain the stream and R-pair levels defining the functions $\times_{str}$ and $\times_R$:

$$\begin{array}{rcl} [\![\times_{str}(x, y)]\!]_{str} & = & ([\![x]\!]_{str} \cdot [\![y]\!]_{str}) \\ [\![\times_R(r, s)]\!]_R & = & ([\![r]\!]_R \cdot [\![s]\!]_R) \end{array}$$

We give below the specification of all the above functions: our multiplication algorithm has quadratic complexity, examining at most the first two digits of its (streams) arguments.

Definition 3.11 ***(Multiplication function)***
The function $\times_{aux} : (str \times str \times str \times [-2,2]) \to str$ *is defined by co-recursion:*

$$\begin{array}{rcl}
\times_{aux}(a : x_0,\ y,\ c : z_0,\ i) & \triangleq & \textbf{\textit{let}}\ (d : e : w) := +_{aux}(\times_{d,str}(a, y),\ z_0,\ i)\ \textbf{\textit{in}} \\
& & \textbf{\textit{let}}\ j := (2d + e + c + i)\ \textbf{\textit{in}} \\
& & \textbf{\textit{Cases}}\ j\ \textbf{\textit{of}} \\
& & \quad j \geq 3\ \Rightarrow\ (1 : \times_{aux}(x_0,\ y,\ w,\ j-4)) \\
& & \quad j \in [-2,2]\ \Rightarrow\ (0 : \times_{aux}(x_0,\ y,\ w,\ j)) \\
& & \quad j \leq -3\ \Rightarrow\ (-1 : \times_{aux}(x_0,\ y,\ w,\ j+4)) \\
& & \textbf{\textit{end}}
\end{array}$$

The function $norm : (Z \times str) \to str$ *is defined by co-recursion:*

$$\begin{array}{rcl}
norm(i,\ a : x_0) & \triangleq & \textbf{\textit{let}}\ j := (2i + a)\ \textbf{\textit{in}} \\
& & \textbf{\textit{Cases}}\ j\ \textbf{\textit{of}} \\
& & \quad j > 0\ \Rightarrow\ (1 : norm(j-1,\ x_0)) \\
& & \quad j = 0\ \Rightarrow\ (0 : norm(0,\ x_0)) \\
& & \quad j < 0\ \Rightarrow\ (-1 : norm(j+1,\ x_0)) \\
& & \textbf{\textit{end}}
\end{array}$$

The function $times_4 : str \to str$ *is defined by:*

$$times_4(a : b : x_1) \triangleq norm(2a + b,\ x_1)$$

The multiplication function on streams $\times_{str} : (str \times str) \to str$ *is defined by:*

$$\times_{str}(x, y) \triangleq times_4(\times_{aux}(x, y, \overline{0}, 0))$$

The multiplication function on R-pairs $\times_R : (R \times R) \to R$ *is defined by:*

$$\times_R(\langle m, x\rangle, \langle n, y\rangle) \triangleq \langle m + n,\ \times_{str}(x, y)\rangle$$

Example 3.1 *Let us exemplify the behavior of the arithmetic functions introduced so far. We detail a very simple example, adding and multiplying the two R-pairs* $1 \triangleq \langle 0, \overline{1}\rangle$ *and* $1/2 \triangleq \langle 0, 1 : \overline{0}\rangle$*. We denote by "*$\triangleright$*" one (lazy) computation step, which is performed in Coq employing suitable tactics, and with* $=_{co}$ *the coinductive, pointwise equality:*

$$\begin{array}{rcl}
+_R(\langle 0, \overline{1}\rangle,\ \langle 0, 1 : \overline{0}\rangle) & \triangleright & \langle 1,\ +_{str}(\overline{1},\ 1 : \overline{0})\rangle \\
& \triangleright & \langle 1,\ +_{aux}(\overline{1},\ \overline{0},\ 2)\rangle \\
& \triangleright & \langle 1,\ 1 : +_{aux}(\overline{1},\ \overline{0},\ 1)\rangle \\
& \triangleright & \langle 1,\ 1 : 1 : +_{aux}(\overline{1},\ \overline{0},\ -1)\rangle \\
& \triangleright & \langle 1,\ 1 : 1 : 0 : +_{aux}(\overline{1},\ \overline{0},\ -1)\rangle \\
& =_{co} & \langle 1,\ 1 : 1 : \overline{0}\rangle
\end{array}$$

$$
\begin{array}{lll}
\times_R(\langle 0,\overline{1}\rangle,\ \langle 0,1:\overline{0}\rangle) & \rhd & \langle 0,\ \times_{str}(\overline{1},\ 1:\overline{0})\rangle \\
 & \rhd & \langle 0,\ times_4(\times_{aux}(\overline{1},\ 1:\overline{0},\ \overline{0},\ 0))\rangle \\
+_{aux}(1:\overline{0},\ \overline{0},\ 0) & =_{co} & 0:1:-1:\overline{0} \\
 & & \\
\times_{aux}(\overline{1},\ 1:\overline{0},\ \overline{0},\ 0) & \rhd & 0:\times_{aux}(\overline{1},\ 1:\overline{0},\ -1:\overline{0},\ 1) \\
+_{aux}(1:\overline{0},\ \overline{0},\ 1) & =_{co} & 1:-1:1:\overline{0} \\
 & & \\
\times_{aux}(\overline{1},\ 1:\overline{0},\ -1:\overline{0},\ 1) & \rhd & 0:\times_{aux}(\overline{1},\ 1:\overline{0},\ 1:\overline{0},\ 1) \\
+_{aux}(1:\overline{0},\ \overline{0},\ 1) & =_{co} & 1:-1:1:\overline{0} \\
 & & \\
\times_{aux}(\overline{1},\ 1:\overline{0},\ 1:\overline{0},\ 1) & \rhd & 1:\times_{aux}(\overline{1},\ 1:\overline{0},\ 1:\overline{0},\ -1) \\
+_{aux}(1:\overline{0},\ \overline{0},\ -1) & =_{co} & 0:-1:1:\overline{0} \\
 & & \\
\times_{aux}(\overline{1},\ 1:\overline{0},\ 1:\overline{0},\ -1) & \rhd & 0:\times_{aux}(\overline{1},\ 1:\overline{0},\ 1:\overline{0},\ -1) \\
 & & \\
\times_{aux}(\overline{1},\ 1:\overline{0},\ \overline{0},\ 0) & =_{co} & 0:0:1:\overline{0} \\
 & & \\
times_4(0:0:1:\overline{0}) & \rhd & norm(0,1:\overline{0}) \\
 & \rhd & 1:norm(0,\overline{0}) \\
 & =_{co} & 1:\overline{0}
\end{array}
$$

Thus $+_R(\langle 0,\overline{1}\rangle,\ \langle 0,1:\overline{0}\rangle) =_{co} \langle 1,1:1:\overline{0}\rangle$ *and* $\times_R(\langle 0,\overline{1}\rangle,\ \langle 0,1:\overline{0}\rangle) =_{co} \langle 0,1:\overline{0}\rangle$. □

Certification of the algorithms. The formal proof that the addition and multiplication functions $+_R$ and $\times_R$ are coherent w.r.t. the corresponding predicates *Add* and *Mult* relies on some preliminary results connecting the auxiliary functions and predicates.

Lemma 3.2 ***(Auxiliary addition functions and predicates)***

$$
\begin{array}{ll}
(i). & \forall x,y \in str,\ \forall a,b,c,d \in \{0,1,-1\},\ \forall i,j \in Z. \\
 & (2a+b+2c+d = 4i+j) \wedge (-2 \leq j \leq 2) \Rightarrow \\
 & add\_aux(a:b:x,\ c:d:y,\ +_{aux}(x,y,j),\ i) \\
 & \\
(ii). & \forall x,y \in str.\ add_{str}(0:x,\ 0:y,\ +_{str}(x,y))
\end{array}
$$

Proof. The lemma is proved formally in the Coq system.

(i). By co-induction.

(ii). Corollary of the point (i): it suffices to consider $x = a : x_0$ and $y = b : y_0$, so $add_{str}(0:x, 0:y, +_{str}(x,y)) = add\_aux(0:a:x_0, 0:b:y_0, +_{aux}(x_0,y_0,a+b), 0)$. □

We have to remember here that the reals are defined by a quotient of a set of representations. We decide to adopt from now on the symbol "$\sim$" for denoting the equivalence on reals that we have introduced in section 3.5. The theorem 3.1 is very useful in the following proofs, because it permits to work inductively or in a co-inductive equivalent setting.

Lemma 3.3 ***(Auxiliary multiplication functions and predicates)***

(i). $\forall x \in str,\ \forall i \in Z.\ less\_aux(norm(i,x),\ x,\ i) \Rightarrow less\_aux(\overline{1},\ x,\ i)$

(ii). $\forall x \in str,\ \forall i \in Z.\ less\_aux(x,\ norm(-i,x),\ i) \Rightarrow less\_aux(x,\ \overline{-1},\ i)$

(iii). $\forall x \in str.$
$\neg less_{str}(x,\ 0:0:\overline{-1}) \wedge \neg less_{str}(0:0:\overline{1},\ x) \Rightarrow$
$\neg less_{str}(x,\ 0:0:times_4(x)) \wedge \neg less_{str}(0:0:times_4(x),\ x)$

(iv). $\forall x, y, z, w \in str.$
$add_{str}(x,y,z) \wedge \neg less_{str}(z,w) \wedge \neg less_{str}(w,z) \Rightarrow$
$add_{str}(x,y,w)$

(v). $\forall x, y, v, w, z \in str,\ \forall d, e \in \{0,1,-1\},\ \forall i, j, k, l \in Z.$
$(-2 \leq i, j \leq 2) \wedge (k + i + 2d + e = 4l + j) \wedge\ add\_aux(v,w,z,k) \Rightarrow$
$add\_aux(\times_{aux}(x,y,v,i),\ d:e:w,\ \times_{aux}(x,y,z,j),\ l)$

(vi). $\forall x, y \in str.\ mult_{str}(x,\ y,\ \times_{str}(x,y))$

Proof. The lemma is proved formally in the Coq system.
(i). By structural induction on $less\_aux(norm(i,x),\ x,\ i)$.
(ii). By structural induction on $less\_aux(x,\ norm(-i,x),\ i)$.
(iii). By points (i) and (ii).
(iv). By definition 3.7 and co-induction.
(v). By lemmas 3.2.(i), 4.3.(v) and 4.3.(vi).
(vi). By points (iii), (iv), (v) and lemma 4.3.(vi). □

We can state and prove now the main result of this section, namely that results computed by the arithmetic functions are admissible for the predicates, and predicates are well-defined with respect to the equivalence.

Proposition 3.4 ***(Arithmetic functions and predicates)***
(i). $\forall r, s \in R.\ Add(r, s, +_R(r,s))$
(ii). $\forall r, s, t, u \in R.\ Add(r,s,t) \wedge Add(r,s,u) \Rightarrow\ (t \sim u)$
(iii). $\forall r, s \in R.\ Mult(r, s, \times_R(r,s))$
(iv). $\forall r, s, t, u \in R.\ Mult(r,s,t) \wedge Mult(r,s,u) \Rightarrow (t \sim u)$

Proof. The proposition is proved formally in the Coq system.
(i). By lemma 3.2.(ii).
(ii). By lemma 4.5.(v) and theorem 4.1.(ii).
(iii). By lemma 3.3.(vi).
(iv). By lemma 4.9.(vi) and theorem 4.1.(ii). □

The proposition establishes that the arithmetic exact algorithms are coherent w.r.t. the specifications given by the predicates. Thus we can derive the properties of the functions from the corresponding properties of the predicates: this is an advantage, because the predicates are easier to work with.

The proposition can also be seen as the formal proof of the reliability of algorithms performing exact real number computation. Thus the code implementing addition and multiplication is *certified*. Using the same approach it is possible to address other more sophisticated functions: we think it would be natural to start from the analytic functions *sin*, *cos*, *exp* and *log* [BC90].

Certified code extraction. One salient feature which distinguishes the proof assistant Coq is the possibility of extracting programs from the constructive content of proofs. This computational interpretation of proof objects fits the tradition of Bishop's constructive mathematics [Bis67] and is based on a realizability interpretation, in the sense of Kleene. This methodology of extracting programs from proofs is a revolutionary paradigm of software engineering, which attracts a continuously growing interest in the theoretical computer science community.

More generally, it is always possible to extract the functional code corresponding to functions specified in Coq. In our case, we are immediately able to obtain the `Haskell` code corresponding to the specifications of addition and multiplication functions. Thus, concerning addition, we are provided with:

```
module Main where

type Prop = () prop = () type Arity = () arity = ()

data nat data treat data str data real

plus n m =
  case n of
    O -> m
    S p -> S (plus p m)
...
r_plus_str x y =
  case x of
    Cons a0 x0 ->
      (case y of
         Cons b0 y0 -> r_plus_aux x0 y0 (zplus (cod a0) (cod b0)))

appendO n x =
  case n of
    O -> x
    S m -> Cons Zero (appendO m x)

r_plus r s =
  case r of
    Pair m x ->
      (case s of
         Pair n y -> Pair (plus (S O) (plus n m))
           (r_plus_str (appendO n x) (appendO m y)))
```

By the validity of proposition 3.4, this code is *certified software*. Alternatively, Coq allows to extract `Objective Caml` and `Scheme` code.

3.7 Related work

We find in the literature several contributions about real numbers in logical frameworks. These works differ each other depending on the fact that the reals can be axiomatized or constructed and on the logical setting used.

The pioneering work is the one of Jutting [Jut77], who used the Automath system [dB70] for the first full-scale attempt to develop and mechanically verify mathematical proofs. From then, several other works have been carried out. The main contribute based on classical logic is probably by Harrison [Har96], who constructs the real numbers in HOL [GM93] by a technique closely related to the Cantor's method, and then develops a significant part of the mathematical analysis, up to integration of functions of a single real variable.

Constructive real numbers, in the Bishop style [Bis67], have been formalized by various authors: Chirimar and Howe [CH92] introduce the reals in the Nuprl system [Con86] and perform a proof of their completeness; Jones [Jon91] uses Lego [Pol94] for studying the completion of general metric spaces; Cederquist [Ced97] follows a point-free topology approach, and uses Half [Mag95] for proving the Hahn-Banach theorem. Other systems used to formalize significant part of the analysis —typically starting from a suitable axiomatization of the real numbers— are Mizar [Rud92], IMPS [FGT90], PVS [ORS92] and Isabelle [Pau94].

The main difference between our approach and the previous ones consists both in the representation chosen for the reals and in the framework we use. Coming specifically to Coq, the proof assistant is equipped with a library `Reals` [May01], which is a classical axiomatization: real numbers are assumed to be a commutative, ordered, Archimedean and complete field. As far as we know, only two constructions of the reals in Coq do exist: the one developed by the FTA [GPWZ00] working group[1], which uses Cauchy sequences, and ours.

In the FTA project, constructive real numbers are just a parameter contextual to a very extensive objective, which concerns the mechanical certification of a theorem of constructive mathematics. Hence the main aim is only to dispose of a suitable axiomatization for describing the reals, such that it is sufficient for proving the Fundamental Theorem of Algebra. We address an articulated discussion about the FTA axiomatization in the next chapter. The formal development produced by the FTA experience has been incorporated in the C-CoRN (The Constructive Coq Repository at Nijmegen), which aims at building a computer-based library of constructive mathematics, formalized in the system Coq.

As remarked above, one model for the FTA axiomatization has been constructed in Coq using the Cauchy completion of the rational numbers [GN01]. Another attempt of the same authors is in progress at time of writing through continued fractions.

[1] The "Fundamental Theorem of Algebra" project - Computing Science Institute, Nijmegen (The Netherlands), 2000.

Chapter 4

Axiomatizations of Constructive Real Numbers

We characterize the Constructive Real Numbers through a minimal axiomatization, that we compare to the alternative proposals of the literature. We prove formally in Coq that our construction of the Reals, carried out in the previous chapter, satisfies our axiomatization.

4.1 A minimal constructive axiomatization

The motivation for introducing a constructive axiomatization of the real numbers is the need of a standard set of properties for proving the adequacy of our construction via co-recursive streams, that we have carried out in the previous chapter. As explained in section 3.3, in the initial phase of our research we looked in the literature for a constructive axiomatization, but we did not find any standard one, so we decided to synthesize a set of axioms by ourselves.

During the progress of our effort we have been aware of the existence of two similar contributions in the literature. One axiomatization is proposed by the FTA group [GPWZ00, GN01] in the context of a very comprehensive project concerning the mechanical certification of the Fundamental Theorem of Algebra in Coq. In that investigation, developed in parallel with respect to ours, the constructive real numbers are just a parameter. An alternative axiomatization, proposed by Bridges [Bri99], is chronologically preceding w.r.t. ours, but we have been aware of it only in the final phase of our synthesis. The main motivation of Bridges coincides only partially with ours: the mathematical curiosity about the properties that are sufficient for characterizing the constructive real numbers in order to develop the constructive real analysis.

The comparison to these two contributions and the very useful and stimulating contacts with their authors have improved the progress of our work. This has allowed a deeper understanding of the constructive real numbers and the statement of a final synthesis. We propose a minimal set of axioms that assume a very small number of primitive notions. This distillation is interesting *per se* for the scientist —the curiosity about an essential characterization of the constructive reals is addressed— and provides a simpler test for possible models. Concerning our co-recursive reals, the minimality shortens and simplifies the task of proving the internal adequacy.

We characterize the constructive real numbers through sixteen axioms organized in four groups: arithmetic operations, ordering, Archimedes' postulate and completeness. Our axiomatization uses only three basic concepts: addition ($+$), multiplication ($\times$) and strict order ($<$). The system we propose is consistent with respect to reference models —(equivalence classes of) Cauchy sequences [TvD88] and co-inductive streams (chapter 3)— and will be compared to the alternative proposals of the literature we have mentioned [Bri99, GN01]. We will prove in particular that our axiomatization has a sufficient deductive power.

We have formalized and used our axioms inside the proof assistant Coq for proving the adequacy of our co-recursive construction of the reals. However, the axioms can be stated and worked with in a general constructive logical setting, because we do not need all the richness of the Calculus of Constructions, the logic beneath Coq. In particular we do not require the use of dependent types and universes. We should only dispose of a logical system that accommodates the second-order quantification —in order to axiomatize the completeness— and the Axiom of Choice —for defining the "reciprocal" function on reals different from zero.

Since most of the constructive approaches to analysis [Bis67, Bee85, Wei00] introduce the real numbers by a quotient of a set of representations (e.g. equivalence classes of Cauchy sequences, digit expansions, etc.), it is necessary to see the reals as a set provided with an equivalence relation. In our proposal, the equivalence ($\sim$) is not a primitive notion, but is derived from the strict order relation; and also its fundamental properties follow from those ones of the strict order. Similarly, it is not necessary to assume the apartness relation ($\#$) —a semi-decidable version of the inequality ($\neq$)— because it is definable in terms of the order relation as well.

Sets, functions, predicates and axioms. We define the constructive real numbers as the mathematical objects satisfying four groups of axioms. The basic notions are the following:

- a representation set R, with two elements 0_R (zero) and 1_R (one);
- a binary relation $<$ (strict order) over R;
- two binary operations $+$ (addition) and $\times$ (multiplication) over R.

We do not assume the existence of the negation ($-$) and reciprocal ($^{-1}$) functions. The main reason for this choice is that the reciprocal function cannot be defined in Coq, because functions have to be totally defined in Coq. Moreover, functions have to be continuous w.r.t. the Euclidean topology in constructive setting, but it is not possible to make continuous by extension the reciprocal function.

In order to state the axioms, it is convenient to define two relations and two functions:

- a binary relation $\sim$ (equivalence) over R tells that two different elements represent the same number, thus capturing the redundancy of the representation;
- two functions $inj : \mathbb{N} \to R$ ($inj(n) = n$) and $exp : \mathbb{N} \to \mathbb{N}$ ($exp(n) = 2^n$) are used in the archimedeanity and completeness axioms;
- a ternary relation $near \subseteq R \times R \times \mathbb{N}$ ($near(x, y, n) \Leftrightarrow |x - y| \leq 2^{-n}$) describes the Euclidean metric.

Our axiomatization is parametric with respect to the set $\mathbb{N}$ of the natural numbers, that we suppose to be given. In our formalization in Coq, $\mathbb{N}$ is taken as the set of the *inductive* natural numbers. In a different context, $\mathbb{N}$ could be defined as a set satisfying the Peano's arithmetic axioms. Finally we claim that constructive real numbers are captured by the following axiomatization.

Definition 4.1 ***(Axioms for constructive real numbers)***

$Consts:$	$R,\ \{0_R, 1_R\} \in R$	$< \subseteq R \times R \quad + : R \times R \to R \quad \times : R \times R \to R$
$Defs:$	$\sim \subseteq R \times R$	$(x \sim y) \triangleq \neg(x < y) \wedge \neg(y < x)$
	$inj : \mathbb{N} \to R$	$inj(0) \triangleq 0_R,\ inj(n+1) \triangleq inj(n) + 1_R$
	$exp : \mathbb{N} \to \mathbb{N}$	$exp(0) \triangleq 1,\ exp(n+1) \triangleq exp(n) \cdot 2$
	$near \subseteq R \times R \times \mathbb{N}$	$near(x, y, n) \triangleq \forall \epsilon \in R.\ (1_R < \epsilon \times inj(exp(n))) \Rightarrow (x < y + \epsilon) \wedge (y < x + \epsilon)$
$Axioms:$	$+$-*associativity*	$\forall x, y, z \in R.\ (x + (y + z)) \sim ((x + y) + z)$
	$+$-*unit*	$\forall x \in R.\ (x + 0_R) \sim x$
	negation	$\forall x \in R.\ \exists y \in R.\ (x + y) \sim 0_R$
	$+$-*commutativity*	$\forall x, y \in R.\ (x + y) \sim (y + x)$
	$\times$-*associativity*	$\forall x, y, z \in R.\ (x \times (y \times z)) \sim ((x \times y) \times z)$
	$\times$-*unit*	$\forall x \in R.\ (x \times 1_R) \sim x$
	reciprocal	$\forall x \in R.\ (0_R < x) \Rightarrow \exists y \in R.\ (x \times y) \sim 1_R$
	$\times$-*commutativity*	$\forall x, y \in R.\ (x \times y) \sim (y \times x)$
	distribuitivity	$\forall x, y, z \in R.\ (x \times (y + z)) \sim (x \times y) + (x \times z)$
	non triviality	$0_R < 1_R$
	$<$ -*asymmetry*	$\forall x, y \in R.\ (x < y) \Rightarrow \neg(y < x)$
	$<$ -*co-transitivity*	$\forall x, y, z \in R.\ (x < y) \Rightarrow (x < z) \vee (z < y)$
	$+$-*reflects-* $<$	$\forall x, y, z \in R.\ (x + z < y + z) \Rightarrow (x < y)$
	$\times$-*reflects-* $<$	$\forall x, y \in R.\ (x \times z < y \times z) \Rightarrow (x < y) \vee ((y < x) \wedge (z < 0_R))$
	archimedeanity	$\forall x \in R.\ \exists n \in \mathbb{N}.\ x < inj(n)$
	completeness	$\forall f : \mathbb{N} \Rightarrow R.\ \exists x \in R.\ (\forall n \in \mathbb{N}.\ near(f(n), f(n+1), n+1)) \Rightarrow (\forall m \in \mathbb{N}.\ near(f(m), x, m))$

□

The minimality of the axiomatization is centered around the characterization of the order, the relationship between the order and the arithmetic operations and the completeness axiom. We are explaining below the four groups of properties.

Arithmetic operations. As the reader can see, the properties required for the arithmetic operations are just the ones characterizing a classical abelian field: in [Bri99] this

set of properties is named "Heyting field". Notice, however, a slight arrangement: it is sufficient to assume the existence of the reciprocal only for positive reals.

As already remarked, we do not assume the negation and the reciprocal functions: instead we assume the existence, for each real x, of its negation element and, if $0 < x$, its reciprocal element. In this way we have to postulate the Axiom of Choice for extracting effectively the negation and the reciprocal of a number x. The necessity of the Axiom of Choice can be seen as a weakness of the axiomatization; however, there is no simple way to avoid it: in fact, without Choice, the reciprocal function cannot be defined in Coq (whereas the negation function and the limit functional could be defined).

An alternative axiomatization not requiring the Axiom of Choice could be obtained as follows. One postulates the existence of the negation and limit functions and, instead of a single inversion function, the existence of a series of approximations of the inversion function, $\mathtt{inv} : (\mathbb{N} \times R) \to R$, satisfying the axiom:

$$\forall n \in \mathbb{N}.\ \forall x \in R.\ (1_R < x \times inj(exp(n))) \Rightarrow (x \times \mathtt{inv}(n, x) \sim 1_R)$$

That is, the function $\lambda x.\ \mathtt{inv}(n, x)$ behaves as the reciprocal function for all the real numbers bigger than 2^{-n}. Given a suitable representation for the reals, the function `inv` can be defined in Coq and allows for the evaluation of the reciprocal of any real number x such that it is possible to find a natural number n satisfying $2^n < x$. We did not pursue this alternative axiomatization for simplicity reasons.

Order relation. First notice that the classical trichotomy of total order $(x < y) \vee (x = y) \vee (y < x)$ fails to be a constructive property [Bri99]: its substitute in the constructive setting is the property $(x < y) \Rightarrow (x < z) \vee (z < y)$, named *<-co-transitivity*.

Next we remark that it is sufficient to define *only* the relation of order, because in constructive mathematics [TvD88, Bri99] the order is universally considered the most fundamental relation for the real numbers. The alternative would be to start from the apartness relation ($\#$) —the constructive inequality ($\neq$)— then to assume axioms for it and further to introduce the order itself with its own axioms [GPWZ00, GN01]. But this choice would cause to add other (redundant) axioms, thus not permitting to carry out our declared purpose to be minimal.

The equivalence ($\sim$) and the apartness ($\#$) relations can be defined using the basic strict order, so their properties are derived from the axioms:

$$(x \sim y) \triangleq \neg(x < y) \wedge \neg(y < x) \qquad (x \,\#\, y) \triangleq (x < y) \vee (y < x)$$

There is still more, because we have been careful in the design of the relationship between the order and the operations. We are able to derive all the basic properties relating the equivalence to the operations from the two reflection axioms:

$$(x + z < y + z) \Rightarrow (x < y)$$
$$(x \times z < y \times z) \Rightarrow (x < y) \vee [(y < x) \wedge (z < 0)]$$

The fact that the equivalence is preserved by the basic notions (order, addition and multiplication) is an immediate consequence of these two axioms and the *<-co-transitivity* one:

$$(x < y) \wedge (x \sim z) \Rightarrow (z < y)$$
$$(x \sim y) \Rightarrow (x + z) \sim (y + z)$$
$$(x \sim y) \Rightarrow (x \times z) \sim (y \times z)$$

Notice that, conversely, the preservation of the equivalence does not follow from the more usual axioms [Bri99, GN01]:

$$(x < y) \Rightarrow (x + z < y + z)$$
$$(0 < x) \wedge (0 < y) \Rightarrow (0 < x \times y)$$

This particular phenomenon relies on the fact that the reflection of the order is more powerful than its preservation, as will be argued in section 4.3.

Archimedeanity. The Archimedean axiom links the real numbers to the natural numbers, stating that the reals are standard with respect to the naturals. The axiom does not exclude the existence of non-standard reals, but in this case also the naturals have to be non-standard. That is, it is possible to conceive non-standard models for our axioms: these models would contain infinitive and infinitesimal real numbers as well as infinitive naturals.

Completeness. The completeness property for the field of the real numbers is postulated asking for the existence of the limit for any Cauchy sequence $\langle s_n \rangle_{n \in \mathbb{N}}$ with an exponential convergence rate:

$$\forall n \in \mathbb{N}.\ |s_n - s_{n+1}| \leq 2^{-(n+1)}$$

Many others choices for capturing the completeness are possible, and our axiom could appear weak at a first glance. Anyway, it is necessary to know the convergence rate of a Cauchy sequence S in order to evaluate constructively its limit: from this convergence rate it is then possible to extract (constructively) a subsequence of S having an exponential convergence rate. Therefore, we are able to derive the completeness properties found in the literature [Bri99, GN01] starting from our axiom. Our choice is motivated by simplicity reasons.

Minimality. Finally, we have characterized the constructive reals as a *commutative, archimedean* and *complete field*, such that the $<$*-co-transitivity* holds. The minimality of the axiomatization is useful both for theoretical reasons and because it provides a simple test for potential models of the reals. Another advantage is the re-usability of proofs: all the properties derived from the axioms are valid for any structure satisfying the same axioms.

It is worth noticing that, for the sake of the clarity of the axiomatization, we have chosen to axiomatize the different notions separately, rather than pursuing a minimal set of axioms at all costs. So we have started from the order, then we have considered the operations and their relationship with the order, and so on. However, if the main aim was to minimize (at all costs) the number of basic notions and axioms, we would have considered a different perspective. For example we could characterize the addition and the order at the same time, using the predicate $(x + y) < (z + w)$; this approach may give an axiomatization even more compact, but be also less natural and comprehensible.

4.2 Consistency of the axioms

We prove in Coq that co-recursive streams, introduced in the previous chapter, are a model for the constructive axiomatization of definition 4.1. The importance of this result

is twofold: we address the internal adequacy of our construction of the reals, and we show that the axioms are inhabited, that is, they are consistent. We present in this section the organization of the formal development in Coq and we discuss the proof techniques used. It is worth noticing that, after some preliminary discussions, the presentation becomes rather technical.

As explained in section 3.2, it is convenient to develop the proofs for the arithmetic operations using predicates and not functions: by proposition 3.4, it is then possible to extend the validity of the theorems to the functions.

Most of the proofs follow a similar pattern: first we prove the basic facts about the auxiliary predicates ($less\_aux$, add_aux), then we deduce the corresponding results about streams ($less_{str}$, add_{str}, $mult_{str}$), and finally we lift the properties to the level of the R-pairs ($Less$, Add, $Mult$). Usually, the main difficulty is to prove the lemma at the "auxiliary" level; the two tactics we have mainly used for mastering this level are `Cofix` and `Omega`.

The tactic `Cofix` is specific for reasoning on co-inductive relations: it is the unique built-in tool for proving co-inductive assertions. It allows to develop top-down infinitely regressive proofs by assuming the conclusion as a premise, provided it is used later only within introduction rules [Coq93, Gim94]. The tactic `Omega` proves automatically assertions in Presburger's arithmetic: it is very useful to avoid repeated case analysis on the values of the ternary digits. The use of this tactic and the introduction of the auxiliary predicates have permitted a great simplification of the proofs: almost all the propositions are proved invoking at most 50 strategies. We detail below the proofs for the different families of axioms.

Order.

Some of the formal proofs use the following induction principle on integer numbers, which is not provided by the standard library of Coq, so has to be added and proved sound:

$$
\begin{array}{l}
P(0) \wedge \\
\forall i \in Z.\ (0 < i) \wedge ((\forall j \in Z.\ (0 \leq j < i) \Rightarrow P(j)) \Rightarrow P(i)) \wedge \\
\forall i \in Z.\ (i < 0) \wedge ((\forall j \in Z.\ (i < j \leq 0) \Rightarrow P(j)) \Rightarrow P(i)) \Rightarrow \\
\forall i \in Z.\ P(i)
\end{array}
$$

Next we need a preparatory lemma about the auxiliary predicate $less\_aux$, that plays a crucial role in the proofs of the axioms. The lemma is used, in turn, for deducing those properties of the predicate $less_{str}$ that have the role of the axioms at the level of streams.

Lemma 4.1 ***(Order: auxiliary level)***
Let be $x, y \in str$, $i, j, k \in Z$ *and* $big = 32$. *Then:*

(i). $less\_aux(x, y, i) \Rightarrow (-1 \leq i)$
(ii). $less\_aux(x, y, i) \wedge (i \leq j) \Rightarrow less\_aux(x, y, j)$
(iii). $(2 < big - i) \Rightarrow less\_aux(x, y, big - i)$
(iv). $(2 < i) \Rightarrow less\_aux(x, y, i)$
(v). $less\_aux(x, y, i) \wedge less\_aux(y, x, j) \Rightarrow (0 < i + j)$
(vi). $less\_aux(x, y, k) \wedge (k \leq i + j) \Rightarrow less\_aux(x, z, i) \vee less\_aux(z, y, j)$

Proof. (i). By structural induction on $less\_aux(x, y, i)$.

(ii). By structural induction on $less\_aux(x, y, i)$.
(iii). By the above induction principle and point (ii).
(iv). Directly by point (iii).
(v). By structural induction on $less\_aux(x, y, i)$ and point (i). The intended meaning is the following: $(\llbracket x \rrbracket_{str} < \llbracket y \rrbracket_{str} + i) \wedge (\llbracket y \rrbracket_{str} < \llbracket x \rrbracket_{str} + j) \Rightarrow (0 < i + j)$.
(vi). By structural induction on $less\_aux(x, y, k)$ and point (iv). The intended meaning is: $(\llbracket x \rrbracket_{str} < \llbracket y \rrbracket_{str} + k) \wedge (k \leq i + j) \Rightarrow (\llbracket x \rrbracket_{str} < \llbracket z \rrbracket_{str} + i) \vee (\llbracket z \rrbracket_{str} < \llbracket y \rrbracket_{str} + j)$. □

Lemma 4.2 ***(Order: stream level)***
Let be $x, y, z \in str$. Then:
(i). $less_{str}(x, y) \Rightarrow \neg less_{str}(y, x)$
(ii). $less_{str}(x, y) \Rightarrow less_{str}(x, z) \vee less_{str}(z, y)$

Proof. (i). By lemma 4.1.(v).
(ii). By lemma 4.1.(vi). □

Finally we can address the axioms of definition 4.1: we establish to represent the two neuter elements $0_R, 1_R$ respectively by the R-pairs $\langle 0,\ \overline{0} \rangle$ and $\langle 0,\ \overline{1} \rangle$.

Theorem 4.1 ***(Order: consistency of the axioms)***
Let r, s, t be R-pairs. Then:
(i). non triviality : $Less(0_R, 1_R)$
(ii). $<$ -asymmetry : $Less(r, s) \Rightarrow \neg Less(s, r)$
(iii). $<$ -co-transitivity : $Less(r, s) \Rightarrow Less(r, t) \vee Less(t, s)$

Proof. (i). By lemma 4.1.(iv).
(ii). By lemma 4.2.(i).
(iii). By lemma 4.2.(ii). □

Addition.

In order to accomplish the *negation* axiom, we have to implement the negation function, working on the concrete structure of R-pairs.

Definition 4.2 ***(Negation function)***
The negation function on streams $-_{str} : str \rightarrow str$ is defined by co-recursion:

$$
\begin{array}{rcl}
-_{str}(a : x) & \triangleq & \textbf{\textit{Cases}}\ a\ \textbf{\textit{of}} \\
 & & \quad 0 \ \Rightarrow\ 0 : (-_{str}(x)) \\
 & & \quad 1 \ \Rightarrow\ -1 : (-_{str}(x)) \\
 & & \quad -1 \ \Rightarrow\ 1 : (-_{str}(x)) \\
 & & \textbf{\textit{end}}
\end{array}
$$

The negation function on R-pairs $-_R : R \rightarrow R$ is defined by:

$$-_R(\langle m, x \rangle) \ \triangleq\ \langle m, -_{str}(x) \rangle$$

Similarly to the case of the order relation, we start by a preparatory lemma about the co-inductive predicate add_aux. The lemma is used in the following for addressing the adequacy of the addition at the level of streams and R-pairs.

Lemma 4.3 ***(Addition: auxiliary level)***
Let be $x, y, z, w_1, w_2, v \in str$, $i, j, k \in Z$ *and* $big = 32$. *Then:*
(i). $add\_aux(x, y, z, i) \wedge (j - big \leq -3) \Rightarrow (j - big \leq i)$
(ii). $add\_aux(x, y, z, i) \Rightarrow (-3 \leq i)$
(iii). $add\_aux(x, y, z, i) \wedge (3 \leq big + j) \Rightarrow (i \leq big + j)$
(iv). $add\_aux(x, y, z, i) \Rightarrow (i \leq 3)$
(v). $add\_aux(x, y, w1, i) \wedge add\_aux(w1, z, v, j) \wedge add\_aux(y, z, w2, i + j - k) \Rightarrow add\_aux(x, w2, v, k)$
(vi). $add\_aux(x, y, z, i) \Rightarrow add\_aux(y, x, z, i)$
(vii). $add\_aux(x, z, w1, i) \wedge add\_aux(y, z, w2, i + j - k) \wedge less\_aux(w1, w2, j) \Rightarrow less\_aux(x, y, k)$.

Proof. (i). By integer induction on j.
(ii). Directly by point (i).
(iii). By integer induction on j.
(iv). Directly by point (iii).
(v). By co-induction and points (ii), (iv). The intended meaning is:

$$\begin{array}{l}(\llbracket x \rrbracket_{str} + \llbracket y \rrbracket_{str} = \llbracket w_1 \rrbracket_{str} + i) \wedge \\ (\llbracket w_1 \rrbracket_{str} + \llbracket z \rrbracket_{str} = \llbracket v \rrbracket_{str} + j) \wedge \\ (\llbracket y \rrbracket_{str} + \llbracket z \rrbracket_{str} = \llbracket w_2 \rrbracket_{str} + (i + j - k)) \Rightarrow \\ (\llbracket x \rrbracket_{str} + \llbracket w2 \rrbracket_{str} = \llbracket v \rrbracket_{str} + k)\end{array}$$

(vi). By co-induction. The intended meaning is:

$$(\llbracket x \rrbracket_{str} + \llbracket y \rrbracket_{str} = \llbracket z \rrbracket_{str} + i) \Rightarrow (\llbracket y \rrbracket_{str} + \llbracket x \rrbracket_{str} = \llbracket z \rrbracket_{str} + i)$$

(vii). By structural induction on $less\_aux(w1, w2, j)$, lemma 4.1.(iv) and points (ii), (iv). The intended meaning is:

$$\begin{array}{l}(\llbracket x \rrbracket_{str} + \llbracket z \rrbracket_{str} = \llbracket w_1 \rrbracket_{str} + i) \wedge \\ (\llbracket y \rrbracket_{str} + \llbracket z \rrbracket_{str} = \llbracket w_2 \rrbracket_{str} + (i + j - k)) \wedge \\ (\llbracket w_1 \rrbracket_{str} < \llbracket w_2 \rrbracket_{str} + j) \Rightarrow \\ (\llbracket x \rrbracket_{str} < \llbracket y \rrbracket_{str} + k)\end{array}$$

□

Lemma 4.4 ***(Addition: stream level)***
Let be $x, y, z, w_1, w_2, v \in str$. *Then:*
(i). $add_{str}(x, y, w1) \wedge add_{str}(w1, z, v) \wedge add_{str}(y, z, w2) \Rightarrow add_{str}(x, w2, v)$
(ii). $add_{str}(\overline{0}, x, x)$
(iii). $add_{str}(x, -_{str}(x), \overline{0})$
(iv). $add_{str}(x, y, z) \Rightarrow add_{str}(y, x, z)$
(v). $add_{str}(x, z, w1) \wedge add_{str}(y, z, w2) \wedge less_{str}(w1, w2) \Rightarrow less_{str}(x, y)$

Proof. (i). By lemma 4.3.(v).
(ii). By co-induction.
(iii). By co-induction.
(iv). By lemma 4.3.(vi).
(v). By lemma 4.3.(vii). □

Lemma 4.5 ***(Addition: R-pair level)***
Let r, s, t, p_1, p_2, q be R-pairs. Then:
(i). $Add(r, s, p_1) \wedge Add(p_1, t, q) \wedge Add(s, t, p_2) \Rightarrow Add(r, p_2, q)$
(ii). $Add(0_R, r, r)$
(iii). $Add(r, -_R(r), 0_R)$
(iv). $Add(r, s, t) \Rightarrow Add(s, r, t)$
(v). $Add(r, t, p_1) \wedge Add(s, t, p_2) \wedge Less(p_1, p_2) \Rightarrow Less(r, s)$

Proof. (i). By lemma 4.4.(i).
(ii). By lemma 4.4.(ii).
(iii). By lemma 4.4.(iii).
(iv). By lemma 4.4.(iv).
(v). By lemma 4.4.(v). □

In order to address the axioms of definition 4.1, we have to prove that the binary predicate "$\sim$", defined in section 3.5 by $(r \sim s) \triangleq \neg Less(r, s) \wedge \neg Less(s, r)$, is an equivalence relation on R-pairs.

Lemma 4.6 ***(Equivalence on R-pairs)***
Let r, s, t be R-pairs. Then:
(i). reflexivity : $(r \sim r)$
(ii). symmetry : $(r \sim s) \Rightarrow (s \sim r)$
(iii). transitivity : $(r \sim s) \wedge (s \sim t) \Rightarrow (r \sim t)$

Proof. (i). By $<$*-asymmetry.*
(ii). Immediate.
(iii). By $<$*-co-transitivity.* □

We can obtain finally the proofs of the axioms: we use below the addition function $+_R$ implemented in definition 3.10.

Theorem 4.2 ***(Addition: consistency of the axioms)***
Let r, s, t be R-pairs. Then:
(i). +-associativity : $(r +_R (s +_R t)) \sim ((r +_R s) +_R t)$
(ii). +-unit : $(r +_R 0_R) \sim r$
(iii). negation : $\exists p \in R.\ (r +_R p) \sim 0_R$
(iv). +-commutativity : $(r +_R s) \sim (s +_R r)$
(v). +-reflects- $<$*:* $Less(r +_R t, s +_R t) \Rightarrow Less(r, s)$

Proof. (i). By proposition 3.4.(i), 3.4.(ii) and lemma 4.5.(i).
(ii). By proposition 3.4.(i), 3.4.(ii) and lemma 4.5.(ii), 4.5.(iv).
(iii). By proposition 3.4.(i), 3.4.(ii) and lemma 4.5.(iii).
(iv). By proposition 3.4.(i), 3.4.(ii) and lemma 4.5.(iv).
(v). By proposition 3.4.(i) and lemma 4.5.(v). □

Multiplication.

Notice that the multiplication predicate is defined in terms of the addition one, and in particular we have avoided defining the "auxiliary" level (see definition 3.6). Thus we

cannot use the `Omega` tactic for carrying out formal proofs about the multiplication. The right technique for approaching the multiplication is to derive first a suite of corollaries for the addition; some of such results are collected in lemma 4.12. The following step is to use these auxiliary properties of addition in the proofs concerning the multiplication.

In order to accomplish the *reciprocal* axiom, we have to introduce the division function. We decide to define only the division between streams, which is used for constructing the reciprocal of R-pairs. Hence we introduce an auxiliary function $test_{str} : (str \times str) \to Z$, used for defining $div_{str} : str \to str$, whose intended meaning is:

$$[\![div_{str}(x, y)]\!]_{str} \quad = \quad [\![x]\!]_{str} / [\![y]\!]_{str}$$

Definition 4.3 ***(Division function)***
The test function on streams $test_{str} : (str \times str) \to Z$ *is defined by:*

$$test_{str}(a : b : c : d : e : x_4,\ f : g : h : y_2) \triangleq 16a + 8b + 4(c - f) + 2(d - g) + (e - h)$$

The division function on streams $div_{str} : str \to str$ *is defined by co-recursion:*

$$\begin{array}{lll}
div_{str}(x,\ y) & \triangleq & \textbf{\textit{let}}\ x := (a : b : c : x_2)\ \textbf{\textit{in}} \\
& & \textbf{\textit{let}}\ j := (4a + 2b + c)\ \textbf{\textit{in}} \\
& & \textbf{\textit{let}}\ i := test(x,\ y)\ \textbf{\textit{in}} \\
& & \textbf{\textit{Cases}}\ j\ \textbf{\textit{of}} \\
& & \quad j \geq 0\ \Rightarrow \quad (*\ [\![x]\!]_{str} \geq -1/8\ *) \\
& & \qquad \textbf{\textit{if}}\ (i \geq 0) \quad (*\ 4[\![x]\!]_{str} - [\![y]\!]_{str} \geq -1/4\ *) \\
& & \qquad \textbf{\textit{then}}\ 1 : div_{str}(times_4(+_{str}(x, -_{str}(0 : y))),\ y) \\
& & \qquad \textbf{\textit{else}}\ 0 : div_{str}(times_4(+_{str}(0 : x, 0 : x)),\ y) \\
& & \quad j < 0\ \Rightarrow \quad (*\ [\![x]\!]_{str} < -1/8\ *) \\
& & \qquad \textbf{\textit{if}}\ (i \geq 0) \quad (*\ 4[\![x]\!]_{str} - [\![y]\!]_{str} \geq -1/4\ *) \\
& & \qquad \textbf{\textit{then}}\ 0 : div_{str}(times_4(+_{str}(0 : x, 0 : x)),\ y) \\
& & \qquad \textbf{\textit{else}}\ -1 : div_{str}(times_4(+_{str}(x, 0 : y)),\ y) \\
& & \textbf{\textit{end}}
\end{array}$$

We need a preparatory lemma relating the multiplication to the addition. The lemma is used after for addressing the adequacy of the multiplication at the level of streams and R-pairs.

Lemma 4.7 ***(Multiplication: auxiliary properties)***
Let be $x, y, z, w_1, w_2, v \in str$, $a, b, c \in \{0, 1, -1\}$, $i, j, k \in Z$, $m \in \mathbb{N}$ *and* $big = 32$. *Then:*
(i). $add\_aux(times_{d,str}(a, \overline{1}), x, x, a)$
(ii). $Less(0_R, \langle m, x\rangle) \Rightarrow \exists y \in str,\ n \in \mathbb{N}.\ \langle m, x\rangle \sim \langle m, (0^n : y)\rangle \wedge leq_{str}(0 : 1 : \overline{0}, y)$
(iii). $mult_{str}(x,\ b : y,\ c : z) \wedge mult_{str}(x, y, w) \Rightarrow add_{str}(0 : times_{d,str}(b, x),\ 0 : w,\ c : z)$
(iv). $add_{str}(x, y, z) \Rightarrow add_{str}(times_{d,str}(a, x), times_{d,str}(a, y), times_{d,str}(a, z))$

Proof. (i). By co-induction.
(ii). Expand the definition of *Less*, then argue by structural induction.
(iii). By co-induction.
(iv). By co-induction. □

Lemma 4.8 ***(Multiplication: stream level)***
Let be $x, y, z, w_1, w_2, u, v \in str$. *Then:*
(i). $mult_{str}(x, y, w1) \wedge mult_{str}(w1, z, v) \wedge mult_{str}(y, z, w2) \Rightarrow mult_{str}(x, w2, v)$
(ii). $mult_{str}(x, \overline{1}, x)$
(iii). $leq_{str}(x, y) \wedge leq_{str}(-y, x) \wedge leq_{str}(0 : 1 : \overline{0}, y) \Rightarrow (\times_{str}(y, div_{str}(x, y)) \sim_{str} x)$
(iv). $mult_{str}(x, y, z) \Rightarrow mult_{str}(y, x, z)$
(v). $add_{str}(y, z, u) \wedge mult_{str}(x, y, w1) \wedge mult_{str}(x, z, w2) \wedge add_{str}(0 : w_1\ , 0 : w_2,\ v) \Rightarrow mult_{str}(x,\ 0 : u,\ v)$
(vi). $mult_{str}(x, z, w1) \wedge mult_{str}(y, z, w2) \wedge less_{str}(w1, w2) \Rightarrow less_{str}(x, y) \vee (less_{str}(y, x) \wedge less_{str}(z, \overline{0}))$

Proof. (i). By co-induction and lemma 3.3.(vi).
(ii). By co-induction and lemma 4.7(i).
(iii). By co-induction.
(iv). By co-induction, lemma 3.3.(vi) and lemma 4.7(iii).
(v). By co-induction, lemma 3.2.(ii) and lemma 4.7(iv).
(vi). By lemma 3.2.(ii), lemma 3.3.(vi) and lemma 4.4(v). □

Lemma 4.9 ***(Multiplication: R-pair level)***
Let r, s, t, o_1, o_2, p, q *be R-pairs. Then:*
(i). $Mult(r, s, p_1) \wedge Mult(p_1, t, q) \wedge Mult(s, t, p_2) \Rightarrow Mult(r, p_2, q)$
(ii). $Mult(r, 1_R, r)$
(iii). $leq_{str}(0 : 1 : \overline{0},\ y) \Rightarrow \times_R(\langle m,\ (0^n : y)\rangle,\ \langle n + 2,\ 0^m : div_{str}(0 : 0 : \overline{1},\ y)\rangle) \sim 1_R$
(iv). $Mult(r, s, t) \Rightarrow Mult(s, r, t)$
(v). $Add(s, t, p) \wedge Mult(r, t, o_1) \wedge Mult(s, t, o_2) \wedge Add(o_1, o_2, q) \Rightarrow Mult(x, p, q)$
(vi). $Mult(r, t, p_1) \wedge Mult(s, t, p_2) \wedge Less(p_1, p_2) \Rightarrow Less(r, s) \vee (Less(s, r) \wedge Less(0_R, t))$

Proof. By lemma 4.8.(i).
(ii). By lemma 4.8.(ii).
(iii).By lemma 4.8.(iii).
(iv). By lemma 4.8.(iv).
(v). By lemma 4.8.(v).
(vi). By lemma 4.8.(vi). □

We can finally obtain the proofs for the axioms: we use below the multiplication function $\times_R$ implemented in definition 3.11.

Theorem 4.3 ***(Multiplication: consistency of the axioms)***
Let r, s, t *be R-pairs. Then:*
(i). $\times$*-associativity :* $(r \times_R (s \times_R t)) \sim ((r \times_R s) \times_R t)$
(ii). $\times$*-unit :* $(r \times_R 1_R) \sim r$
(iii). reciprocal : $Less(0_R, r) \Rightarrow \exists p \in R.\ (r \times_R p) \sim 1_R$
(iv). $\times$*-commutativity :* $(r \times_R s) \sim (s \times_R r)$
(v). distribuitivity : $(r \times_R (s +_R t)) \sim (r \times_R s) +_R (r \times_R t)$
(vi). $\times$*-reflects-* $<$*:* $Less(r \times_R t, s \times_R t) \Rightarrow Less(r, s) \vee ((Less(s, r) \wedge Less(t, 0_R))$

Proof. (i). By lemma 4.9.(i).
(ii). By lemma 4.9.(ii).
(iii). Let be $r \equiv \langle m, x\rangle$; by lemma 4.7.(ii) there exist n, y such that $\langle m, x\rangle \sim \langle m, 0^n : y\rangle$ and $leq_{str}(0 : 1 : \overline{0},\ y)$. Pick out $p := \langle n+2,\ 0^m : div_{str}(0 : 0 : \overline{1}, y)\rangle$, thus concluding by lemma 4.9.(iii).
(iv). By lemma 4.9.(iv).
(v). By lemma 4.9.(v).
(vi). By lemma 4.9.(vi). □

Archimedeanity.

It is quite simple to show that the Archimedean axiom is satisfied by co-inductive streams: given the R-pair $\langle n, x\rangle$, it is actually sufficient to choose the witness 2^{n+1}. In order to adhere to the statement of the axiomatization 4.1, it is necessary to implement the injection and exponential functions.

Definition 4.4 ***(Injection, exponential)***
The injection function $inj_R : \mathbb{N} \to R$ is defined by recursion:

$$\begin{array}{rcl} inj_R(n) & \triangleq & \textbf{\textit{Cases}}\ n\ \textbf{\textit{of}} \\ & & \quad 0 \ \Rightarrow\ 0_R \\ & & \quad m+1 \ \Rightarrow\ +_R(inj_R(m), 1_R) \\ & & \textbf{\textit{end}} \end{array}$$

The exponential function on naturals $exp_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}$ is defined by recursion:

$$\begin{array}{rcl} exp_{\mathbb{N}}(n) & \triangleq & \textbf{\textit{Cases}}\ n\ \textbf{\textit{of}} \\ & & \quad 0 \ \Rightarrow\ 1 \\ & & \quad m+1 \ \Rightarrow\ 2 * exp_{\mathbb{N}}(m) \\ & & \textbf{\textit{end}} \end{array}$$

We need a preparatory lemma about the functions inj_R and $exp_{\mathbb{N}}$, which allows to address the consistency of the Archimedeanity axiom.

Lemma 4.10 ***(Archimedeanity: auxiliary properties)***
Let be $x \in str$, $m, n \in \mathbb{N}$. Then:
(i). $inj_R(m+n) \sim inj_R(m) +_R inj_R(n)$
(ii). $Less(\langle n, x\rangle,\ \langle n+1, \overline{1}\rangle)$
(iii). $\langle m, x\rangle \sim \langle n+m, 0^n : x\rangle$
(iv). $\langle n+1, \overline{1}\rangle \sim \langle n, \overline{1}\rangle +_R \langle n, \overline{1}\rangle$
(v). $\langle n, \overline{1}\rangle \sim inj_R(exp_{\mathbb{N}}(n))$

Proof. (i). By induction on m and lemma 4.12.(2).
(ii). By induction on n and lemma $\forall x.\ less\_str(0 : x, \overline{1})$, which is proved trivially.
(iii). Immediate by definition.
(iv). By point (iii).
(v). By induction on n and (i). □

Theorem 4.4 ***(Archimedeanity: consistency of the axiom)***
Let be r an R-pair. Then $\exists n \in \mathbb{N}$ such that $Less(r, inj_R(n))$.

Proof. Let be $r \equiv \langle m, x\rangle$: choose $n := exp_{I\!N}(m+1)$ and conclude by lemma 4.10, points (iv) and (v), and lemma 4.11.(3). □

Completeness.

In order to accomplish the *completeness* axiom, we have to introduce some auxiliary functions working on streams and R-pairs. First we need the addition and subtraction functions $plus_1 : str \to str$ and $minus_1 : str \to str$, whose intended meaning is the following:

$$[\![plus_1(x)]\!]_{str} = \begin{cases} [\![x]\!]_{str} + 1 & \text{if } [\![x]\!]_{str} + 1 \leq 1 \\ (+1) & \text{if } [\![x]\!]_{str} + 1 > 1 \end{cases}$$

$$[\![minus_1(x)]\!]_{str} = \begin{cases} [\![x]\!]_{str} - 1 & \text{if } [\![x]\!]_{str} - 1 \geq -1 \\ (-1) & \text{if } [\![x]\!]_{str} - 1 < -1 \end{cases}$$

Next we introduce a multiplication and division functions $times_{exp} : (I\!N \times str) \to str$ and $div_{exp} : (I\!N \times str) \to str$:

$$[\![times_{exp}(n, x)]\!]_{str} = \begin{cases} [\![x]\!]_{str} \cdot 2^n & \text{if } [\![x]\!]_{str} \cdot 2^n \leq 1 \\ (+1) & \text{if } [\![x]\!]_{str} \cdot 2^n > 1 \end{cases}$$

$$[\![div_{exp}(n, x)]\!]_{str} = [\![x]\!]_{str} \cdot 2^{-n}$$

We employ two normalization functions for the construction of the limit: $norm_{str} : (digit \times str) \to str$ constraints the first digit of a stream, thus transforming accordingly its tail; $norm_R : (I\!N \times R) \to str$ constraints the exponent of an R-pair, operating coherently on the mantissa:

$$[\![norm_{str}(a, x)]\!]_{str} = \begin{cases} 2 \cdot [\![x]\!]_{str} - a & \text{if } -1 \leq (2 \cdot [\![x]\!]_{str} - a) \leq 1 \\ (+1) & \text{if } (2 \cdot [\![x]\!]_{str} - a) > 1 \\ (-1) & \text{if } (2 \cdot [\![x]\!]_{str} - a) < -1 \end{cases}$$

$$[\![norm_R(n, r)]\!]_R = [\![r]\!]_R \cdot 2^{-n}$$

The formalization of all the above functions is as follows.

Definition 4.5 ***(Limit auxiliary functions)***
The functions $plus_1 : str \to str$ and $minus_1 : str \to str$ are defined by co-recursion:

$$\begin{aligned} plus_1(a : x) \triangleq\ & \textbf{\textit{Cases}}\ a\ \textbf{\textit{of}} \\ & \quad 1 \Rightarrow \overline{1} \\ & \quad 0 \Rightarrow 1 : plus_1(x) \\ & \quad -1 \Rightarrow 1 : x \\ & \textbf{\textit{end}} \end{aligned}$$

$$\begin{aligned} minus_1(a : x) \triangleq\ & \textbf{\textit{Cases}}\ a\ \textbf{\textit{of}} \\ & \quad 1 \Rightarrow -1 : x \\ & \quad 0 \Rightarrow -1 : minus_1(x) \\ & \quad -1 \Rightarrow \overline{-1} \\ & \textbf{\textit{end}} \end{aligned}$$

The functions $times_{exp} : (I\!N \times str) \to str$ *and* $div_{exp} : (I\!N \times str) \to str$ *are defined by recursion:*

$$\begin{array}{lcl} times_{exp}(n,\ a:x) & \triangleq & \textbf{\textit{Cases}}\ n\ \textbf{\textit{of}} \\ & & \quad 0\ \Rightarrow\ a:x \\ & & \quad m+1\ \Rightarrow\ times_{exp}(m,\ norm_{str}(a,x)) \\ & & \textbf{\textit{end}} \end{array}$$

$$\begin{array}{lcl} div_{exp}(n,x) & \triangleq & \textbf{\textit{Cases}}\ n\ \textbf{\textit{of}} \\ & & \quad 0\ \Rightarrow\ x \\ & & \quad m+1\ \Rightarrow\ 0 : div_{exp}(m,x) \\ & & \textbf{\textit{end}} \end{array}$$

The functions $norm_{str} : (digit \times str) \to str$ *and* $norm_R : (I\!N \times R) \to str$ *are defined by:*

$$\begin{array}{lcl} norm_{str}(a,\ b:x) & \triangleq & \textbf{\textit{let}}\ j := (b-a)\ \textbf{\textit{in}} \\ & & \textbf{\textit{Cases}}\ j\ \textbf{\textit{of}} \\ & & \quad j=0\ \Rightarrow\ x \\ & & \quad j>0\ \Rightarrow\ plus_1(x) \\ & & \quad j<0\ \Rightarrow\ minus_1(x) \\ & & \textbf{\textit{end}} \end{array}$$

$$\begin{array}{lcl} norm_R(n,\ \langle m,x\rangle) & \triangleq & \textbf{\textit{let}}\ j := (n-m)\ \textbf{\textit{in}} \\ & & \textbf{\textit{Cases}}\ j\ \textbf{\textit{of}} \\ & & \quad j\leq 0\ \Rightarrow\ times_{exp}(-j,\ x) \\ & & \quad j>0\ \Rightarrow\ div_{exp}(j,\ x) \\ & & \textbf{\textit{end}} \end{array}$$

The algorithm for the construction of the limit works as follows. Given a sequence $\langle s_n\rangle_{n\in I\!N}$ satisfying the premise of the completeness axiom:

$$\forall n \in I\!N.\ |s_n - s_{n+1}| \leq 2^{-(n+1)}$$

we consider the subsequence $\langle t_n\rangle_{n\in I\!N} = \langle s_{n+3}\rangle_{n\in I\!N}$. Let be $t_0 = \langle m,x\rangle$: we construct a sequence $\langle u_n\rangle_{n\in I\!N}$ such that $[\![u_n]\!]_R = [\![t_n]\!]_R$ for every $n \in I\!N$ and the exponent of every u_n is $m+1$. This is carried out using the function $norm_R$. The exponent of the limit is $m+1$, while the mantissa is obtained from the sequence $\langle x_n\rangle_{n\in I\!N}$, where $\langle u_n\rangle_{n\in I\!N} = \langle m+1, x_n\rangle_{n\in I\!N}$. The limit of the stream sequence $\langle x_n\rangle_{n\in I\!N}$ is constructed by generating the first digit looking at the first three digits of x_0 and applying co-recursively the method to the subsequence $\langle x_{i+1}\rangle_{i\in I\!N}$, point-wise modified by the function $norm_{str}$. This machinery requires to introduce the following functions.

Definition 4.6 ***(Limit functions)***
The limit function on stream sequences $lim_{str} : (I\!N \to str) \to str$ *is defined by co-*

recursion:

$$\begin{array}{lcl} lim_{str}(\langle x_n\rangle_{n\in\mathbb{N}}) & \triangleq & \textbf{\textit{let}}\ x_0 := (a:b:c:y)\ \textbf{\textit{in}} \\ & & \textbf{\textit{let}}\ j := (4a+2b+c)\ \textbf{\textit{in}} \\ & & \textbf{\textit{Cases}}\ j\ \textbf{\textit{of}} \\ & & \quad j \in [3,7]\ \Rightarrow\ 1 : (lim_{str}(\lambda n.\ (norm_{str}(1,\ x_{n+1})))) \\ & & \quad j \in [-2,2]\ \Rightarrow\ 0 : (lim_{str}(\lambda n.\ (norm_{str}(0,\ x_{n+1})))) \\ & & \quad j \in [-7,-3]\ \Rightarrow\ -1 : (lim_{str}(\lambda n.\ (norm_{str}(-1,\ x_{n+1})))) \\ & & \textbf{\textit{end}} \end{array}$$

The pre-normalization function on R-pair sequences $prenorm : (\mathbb{N} \to R) \to (\mathbb{N} \to str)$ *is defined by:*

$$\begin{array}{lcl} prenorm(\langle r_n\rangle_{n\in\mathbb{N}}) & \triangleq & \textbf{\textit{let}}\ r_0 := \langle m, x\rangle\ \textbf{\textit{in}} \\ & & \lambda n.\ lim_{str}(norm_R(m+1,\ r_n)) \end{array}$$

The auxiliary limit function on R-pair sequences $lim_R : (\mathbb{N} \to R) \to R$ *is defined by:*

$$\begin{array}{lcl} lim_R(\langle r_n\rangle_{n\in\mathbb{N}}) & \triangleq & \textbf{\textit{let}}\ r_0 := \langle m, x\rangle\ \textbf{\textit{in}} \\ & & \langle m+1,\ prenorm(\langle r_n\rangle_{n\in\mathbb{N}})\rangle \end{array}$$

The limit function on R-pair sequences $lim : (\mathbb{N} \to R) \to R$ *is defined by:*

$$lim(\langle r_n\rangle_{n\in\mathbb{N}}) \triangleq lim_R(\lambda n.\ \langle r_{n+3}\rangle_{n\in\mathbb{N}})$$

We need to address formally the properties of all the functions we have introduced so far: these results allow to establish the consistency of the completeness axiom.

Theorem 4.5 ***(Completeness: consistency of the axiom)*** *Let* $\langle r_n\rangle_{n\in\mathbb{N}}$ *be a sequence of R-pairs such that, for all* $n \in \mathbb{N}$. $near(r_n, r_{n+1}, n+1)$. *Then, for all* $m \in \mathbb{N}$. $near(r_m, lim(\langle r_n\rangle_{n\in\mathbb{N}}), m)$.

Proof. Expand the hypothesis, then use the properties of the involved functions. □

Example 4.1 *We can use our limit construction algorithm for dealing with the real, irrational number* π. *Many constructions of* π *can be found in the literature; one of the simpler ones is due to Leibniz:*

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots$$

Thus we can immediately define a sequence of rational numbers converging to π. *Let be:*

$$\begin{array}{lcl} f(0) & \triangleq & 4 \\ f(n+1) & \triangleq & f(n) + 4(-1)^n/(2n+1) \\ g(n) & \triangleq & f(2^n) \end{array}$$

Hence $\pi \triangleq lim(\langle g_n\rangle_{n\in\mathbb{N}})$. □

Internal adequacy/Consistency of the Axiomatization.

We collect all the results of this section in the following ultimate proposition.

Proposition 4.1 *Our construction of the real numbers satisfies the axiomatization 4.1.* □

The formal development we have described is documented at the web page of the author[1].

[1] `http://www.dimi.uniud.it/~ciaffagl`

4.3 Axioms at work

In this section we point out the elementary mathematical theory arising from the axiomatization 4.1, following [CDG01]. The results we derive provide a richer package of properties, useful for deducing, in turn, more advanced properties of the constructive real numbers.

It is worth noticing that all the proofs we argue are based on constructive logic and have been carried out formally in the proof assistant Coq. We begin with the order and involved notions; let be $r, s \in R$; we remember the following definitions:

$$\begin{array}{lcl} (r \sim s) & \triangleq & \neg(r < s) \wedge \neg(s < r) \\ (r \# s) & \triangleq & (r < s) \vee (s < r) \\ (r \leq s) & \triangleq & \neg(s < r) \end{array}$$

Using the $<$-*asymmetry* and the $<$-*co-transitivity* axioms we have immediately that $\sim$ is an equivalence relation. Thus we can establish our first result.

Lemma 4.11 ***(Order: derived properties)***
The following properties of the order follow from the axiomatization 4.1:

(1). irreflexivity: $\forall x.\ \neg(x < x)$
(2). transitivity: $\forall x, y, z.\ (x < y) \wedge (y < z) \Rightarrow (x < z)$
(3). preservation by equivalence: $\forall x, y, z, w.\ (x < y) \wedge (x \sim z) \wedge (y \sim w) \Rightarrow (z < w)$
(4). co-transitivity of apartness: $\forall x, y, z.\ (x \# y) \Rightarrow (x \# z) \vee (z \# y)$
(5). apartness preservation: $\forall x, y, z, w.\ (x \# y) \wedge (x \sim z) \wedge (y \sim w) \Rightarrow (z \# w)$

Proof. Point (1) is just an instance of the $<$-*asymmetry*.

(2). We deduce $(x < z) \vee (z < y)$ from $(x < y)$ using the $<$-*co-transitivity*: the left case is the thesis; the right alternative is in contradiction with the hypothesis $(y < z)$.

(3). We derive $(x < z) \vee (z < y)$ from $(x < y)$ by $<$-*co-transitivity*. The case $(x < z)$ contradicts the second hypothesis $(x \sim z)$. By applying again the $<$-*co-transitivity* to the alternative $(z < y)$, we obtain $(z < w) \vee (w < y)$, but $(w < y)$ is in contradiction with the hypothesis $(y \sim w)$.

(4). An immediate implication of the $<$-*co-transitivity*.

(5). We have $(x \# z) \vee (z \# y)$ from $(x \# y)$ by point (4). The left case contradicts the hypothesis $(x \sim z)$. We deduce the thesis from the right case $(z \# y)$ by the point (4). □

We state the main fact concerning the addition, i.e. it preserves the equivalence. We deduce also the preservation of the order, which is postulated as axiom in the alternative axiomatizations [Bri99, GN01].

Lemma 4.12 ***(Addition: derived properties)***
The following properties of the addition follow from the axiomatization 4.1:

(1). strong extensionality : $\forall x, y, z, w.\ (x + y) \# (z + w) \Rightarrow (x \# z) \vee (y \# w)$
(2). equivalence preservation : $\forall x, y, z, w.\ (x \sim y) \wedge (z \sim w) \Rightarrow (x + z) \sim (y + w)$
(3). order preservation : $\forall x, y, z.\ (x < y) \Rightarrow (x + z < y + z)$
(4). equivalence reflection : $\forall x, y, z.\ (x + z) \sim (y + z) \Rightarrow (x \sim y)$

Proof. (1). The goal follows immediately from the two judgments $(x+y)\,\#\,(z+y) \Rightarrow (x\,\#\,z)$ (left extensionality) and $(x+y)\,\#\,(x+z) \Rightarrow (y\,\#\,z)$ (right extensionality). Left extensionality can be written as $(x+y<z+y) \vee (z+y<x+y) \Rightarrow (x<z) \vee (z<x)$ and proved by cases using the *+-reflects-<* axiom. Right extensionality can be reduced to left extensionality by *co-transitivity of apartness*, *apartness preservation* and the *+-commutativity* axiom.

(2). Immediate by point (1), because $\forall r, s \in R.\ (r \sim s) \Leftrightarrow \neg(r\,\#\,s)$.

(3). Using *opposite*, point (2), *+-unit* and *+associativity* we can derive $x \sim (x+z)+(-z)$ and $y \sim (y+z)+(-z)$. By lemma 4.11.(3) it is then possible to deduce $(x+z)+(-z) < (y+z)+(-z)$, and from this the thesis via *+-reflects-<*.

(4). By point (3) we have $(x\,\#\,y) \Rightarrow (x+z)\,\#\,(y+z)$, and thus the thesis. □

We obtain analogous results for the multiplication: we prove it preserves the equivalence and additional properties.

Lemma 4.13 ***(Multiplication: derived properties)***
The following properties of the multiplication follow from the axiomatization 4.1:

(1). strong extensionality: $\forall x, y, z, w.\ (x \times y)\,\#\,(z \times w) \Rightarrow (x\,\#\,z) \vee (y\,\#\,w)$
(2). equivalence preservation: $\forall x, y, z, w.\ (x \sim y) \wedge (z \sim w) \Rightarrow (x \times z) \sim (y \times w)$
(3). positivity reflection: $\forall x, y, z.\ (x \times z) < (y \times z) \wedge (0 < z) \Rightarrow (x < y)$
(4). zero annuls multiplication: $\forall x.\ (x \times 0 \sim 0)$
(5). reciprocal preserves positivity: $\forall x.\ (0 < x) \Rightarrow (0 < x^{-1})$
(6). positivity preservation: $\forall x, y.\ (0 < x) \wedge (0 < y) \Rightarrow (0 < x \times y)$

Proof. (1), (2). By the same arguments used in the corresponding proofs for the addition.

(3). By axiom *×-reflects-<*.

(4). From $(0+0) \sim 0$, using point (2), *distributivity*, *+-unit* and *+-commutativity* we derive $(x \times 0)+(x \times 0) \sim (0+(x \times 0))$: the thesis follows through the lemma 4.12.(4).

(5). From $(0 < x)$, *inverse*, *×-commutativity* and *non triviality* we obtain $0 < (x^{-1} \times x)$; then, by point (4), *×-commutativity* and *preservation by equivalence* we have $(0 \times x) < (x^{-1} \times x)$, hence we conclude by point (3).

(6). Using *×-unit*, *inverse*, point (2) and *×-associativity* we derive $x \sim ((x \times y) \times y^{-1})$, from which $(0 \times y^{-1}) < ((x \times y) \times y^{-1})$: we deduce the thesis by points (5) and (3). □

It is easy to prove that the *+-reflects-<* axiom is equivalent to the *equivalence preservation* plus the *order preservation* of lemma 4.12. In [Bri99] and [GN01] these two properties are taken as axioms: we prefer our choice for minimality reasons. A similar argument applies to the multiplication.

We remark also that the following alternative candidates for the *+-reflects-<* axiom are too weak:

$$(x \times z < y \times z) \wedge (0 < z) \Rightarrow (x < y)$$
$$(x \times z < y \times z) \Rightarrow (x < y) \vee (z < 0)$$

We collect other useful properties of the constructive real numbers that involve the derived non-strict order $(\leq)$ and the apartness $(\#)$.

Lemma 4.14 ***(Other properties)***
The following judgments are derived from the axiomatization 4.1 and lemmas 4.11, 4.12, 4.13:

Order :
1. $(x \leq y) \wedge (y \leq x) \Rightarrow (x \sim y)$
2. $(x \leq y) \wedge (y \leq z) \Rightarrow (x \leq z)$
3. $(x \leq y) \wedge (y < z) \Rightarrow (x < z)$
4. $(x \leq y) \wedge (z < x) \Rightarrow (z < y)$
5. $((z < x) \Rightarrow (z < y)) \Rightarrow (x \leq y)$
6. $(x < y) \vee (x \sim y) \Rightarrow (x \leq y)$

Addition :
7. $(0 < x) \Rightarrow (-x < 0)$
8. $(0 < x) \wedge (0 < y) \Rightarrow (0 < x + y)$
9. $(0 < x + y) \Rightarrow (0 < x) \vee (0 < y)$
10. $(x \leq y) \Leftrightarrow (x + z \leq y + z)$

Multiplication :
11. $(x \times (-y)) \sim -(x \times y)$
12. $(x < y) \wedge (0 < z) \Rightarrow (x \times z < y \times z)$
13. $(x < y) \wedge (z < 0) \Rightarrow (y \times z < x \times z)$
14. $(x \times y < 0) \Rightarrow (x < 0) \vee (y < 0)$
15. $(0 < x \times y) \Rightarrow (x \# 0) \wedge (y \# 0)$
16. $(0 \leq x \times x)$
17. $(x \# 0) \Leftrightarrow (0 < x \times x)$

Proof. (1). A simple consequence of the definitions.
(2), (3), (4). By $<$-*co-transitivity*.
(5). By $<$-*co-transitivity* and 4.11.(1).
(6). By $<$-*asymmetry* and definition.
(7). By 4.12.(3), $+$-*unit*, *opposite* and 4.11.(3).
(8). By 4.12.(3) and 4.11.(2).
(9). By $<$-*co-transitivity*, 4.12.(3), $+$-*associativity*, $+$-*unit*, *opposite*, 4.12.(2) and 4.11.(3).
(10). By $+$-*reflects*-$<$ for $\Rightarrow$; by 4.12.(3) for $\Leftarrow$.
(11). By *opposite*, 4.13.(2), *distributivity*, 4.13.(4), $\times$-*commutativity* and 4.12.(4).
(12). By 4.12.(3), *opposite*, $\times$-*associativity*, $\times$-*commutativity*, 4.11.(3), 4.13.(6) and *distribuitivity*.
(13). By (7), (12), (11), 4.11.(3), 4.12.(3), *opposite*, $\times$-*commutativity* and $\times$-*associativity*.
(14). By 4.13.(4), 4.11.(3) and $\times$-*reflects*-$<$.
(15). By 4.13.(4), 4.11.(3), $\times$-*reflects*-$<$ and 4.13.(3).
(16). By 4.13.(4), 4.11.(3), $\times$-*reflects*-$<$ and (13).
(17). By (13) and (12) for $\Rightarrow$; by 4.13.(4) and $\times$-*reflects*-$<$ for $\Leftarrow$. □

4.4 Equivalent axiomatizations

In this section we compare the axiomatization 4.1 —that we have proposed for characterizing the constructive real numbers— with other axiomatizations in the literature

[TvD88, Bri99, GN01]: we prove in particular that our axiomatization has the same deductive power of the alternative ones. The proof is carried out in the proof assistant Coq.

The work of Troelstra and van Dalen [TvD88] provides a constructive treatment of the theory of the real numbers in the framework of constructive mathematics. Even if the authors do not address explicitly the quest for an axiomatization, their approach focus on aspects strictly related to the present work. Troelstra and van Dalen build the reals by equivalence classes of fundamental (Cauchy) sequences of rationals; then they define the primitive strict order relation ($<$) and the arithmetic functions ($+, \cdot$) on reals. The basic properties of these notions are the ones we have proved in section 4.3, or follow simply from those results.

We mainly refer our work to the other two contributions: the FTA working group[2] and Bridges [Bri99] address explicitly the synthesis of a constructive axiomatization of the reals.

FTA's axiomatization.

The approach used in the FTA project [GPWZ00] is similar to ours in regard to the tool used for the formalization, i.e. the proof assistant Coq. Anyway, in FTA, the constructive real numbers are just a parameter contextual to a very extensive objective, which concerns the mechanical certification of a theorem of constructive mathematics. Hence the aim is only to dispose of a set of properties for describing the reals such that it is sufficiently powerful for proving the Fundamental Theorem of Algebra.

The FTA axiomatization is used also for addressing the adequacy of a construction of the reals through Cauchy sequences [GN01], a contribute we have already mentioned as related work in section 3.7. The FTA axiomatization is developed as follows: first the algebraic structure of *constructive setoid* is introduced via an *apartness* relation; successively the notion of constructive real number is gained step by step. We are proving the equivalence between our axiomatization and the FTA one; we remark that the proof is carried out formally in Coq.

The FTA approach uses 28 axioms. The essential differences with respect to ours are the introduction of the apartness relation as primitive, the assumption of strongly extensionality for the arithmetic functions and the statement of the completeness axiom.

The structure. Constructive reals are defined by the tuple:

$$\langle I\!R, 0, 1, +, *, -, ^{-1}, =, <, \# \rangle$$

Field. The following axioms characterize the notion of constructive setoid:

$$\begin{array}{l}
\texttt{ap_irr} : \forall x.\ \neg(x \# x) \\
\texttt{ap_sym} : \forall x, y.\ (x \# y) \Rightarrow (y \# x) \\
\texttt{ap_cot} : \forall x, y.\ (x \# y) \Rightarrow \forall z.\ (x \# z) \vee (z \# y) \\
\texttt{ap_tight} : \forall x, y.\ \neg(x \# y) \Leftrightarrow (x = y)
\end{array}$$

[2]The "Fundamental Theorem of Algebra" project - Computing Science Institute, Nijmegen (The Netherlands), 2000

Some of the properties leading to a constructive field coincide with ours:

$$\begin{aligned}
&\texttt{add_assoc} : \forall x, y, z.\ (x + (y + z)) = ((x + y) + z)\\
&\texttt{add_unit} : \forall x.\ (x + 0) = x\\
&\texttt{add_commut} : \forall x, y.\ (x + y) = (y + x)\\
&\texttt{minus_proof} : \forall x.\ x + (-x) = 0\\
&\texttt{mult_assoc} : \forall x, y, z.\ (x * (y * z)) = ((x * y) * z)\\
&\texttt{mult_unit} : \forall x.\ (x * 1) = x\\
&\texttt{mult_commut} : \forall x, y.\ (x * y) = (y * x)\\
&\texttt{dist} : \forall x, y, z.\ (x * (y + z)) = (x * y) + (x * z)\\
&\texttt{rcpcl_proof} : \forall x.\ (x \# 0) \Rightarrow (x * (x^{-1})) = 1
\end{aligned}$$

FTA uses also the extra axioms:

$$\begin{aligned}
&\texttt{add_strext} : \forall x, y, z, u.\ (x + y) \# (z + u) \Rightarrow (x \# z) \vee (y \# u)\\
&\texttt{minus_strext} : \forall x, y.\ (-x) \# (-y) \Rightarrow (x \# y)\\
&\texttt{mult_strext} : \forall x, y, z, u.\ (x * y) \# (z * u) \Rightarrow (x \# z) \vee (y \# u)\\
&\texttt{non_triv} : 1 \# 0\\
&\texttt{rcpcl_ap_zero} : \forall x.\ (x \# 0) \Rightarrow (x^{-1} \# 0)\\
&\texttt{rcpcl_strext} : \forall x, y.\ (x^{-1}) \# (y^{-1}) \Rightarrow (x \# y)
\end{aligned}$$

Ordered field. The only axiom shared with ours is the asymmetry:

$$\texttt{less\_asym} : \forall x, y.\ (x < y) \Rightarrow \neg(y < x)$$

The extra axioms are the following:

$$\begin{aligned}
&\texttt{less_strext} : \forall x, y, z, u.\ (x < y) \Rightarrow (z < u) \vee (x \# z) \vee (y \# u)\\
&\texttt{less_trans} : \forall x, y, z.\ (x < y) \wedge (y < z) \Rightarrow (x < z)\\
&\texttt{less_irr} : \forall x.\ \neg(x < x)\\
&\texttt{add_resp_less} : \forall x, y, z.\ (x < y) \Rightarrow (x + z < y + z)\\
&\texttt{times_resp_pos} : \forall x, y.\ (0 < x) \wedge (0 < y) \Rightarrow (0 < x * y)\\
&\texttt{less_conf_ap} : \forall x, y, z.\ (x \# y) \Leftrightarrow (x < y) \vee (y < x).
\end{aligned}$$

Archimedeanity. This axiom coincides with ours. Let be F a field; then:

$$\texttt{arch\_proof} : \forall x \in F.\ \exists n \in I\!N.\ (x < \texttt{nreal}(n))$$

where the function *nreal* is an injection of the natural numbers into the reals.

Completeness. The axiom asks that every Cauchy sequence has a limit. Let be F a field; then:

$$\begin{aligned}
\texttt{lim_proof} :\ & \forall s : I\!N \to F,\ \forall c : \texttt{cauchy}(s).\ \forall \epsilon \in F.\ (0 < \epsilon) \Rightarrow \exists n \in I\!N.\\
& \forall m \in I\!N.\ (n \leq m) \Rightarrow |\, s(m) - \texttt{lim}(s, c)\,| < \epsilon
\end{aligned}$$

The function *lim* takes a Cauchy sequence and returns its limit; *cauchy* is a formalization of the well-known condition:

$$\begin{aligned}
\texttt{lim} :\ & \forall s : I\!N \to I\!R.\ \texttt{cauchy}(s) \to I\!R\\
\texttt{cauchy} :\ & \forall f : I\!N \to F.\ \forall \epsilon \in F.\ (0 < \epsilon) \Rightarrow \exists n \in I\!N.\\
& \forall m \in I\!N.\ (n \leq m) \Rightarrow |\, f(m) - f(n)\,| < \epsilon
\end{aligned}$$

We prove that the different and extra axioms used by FTA can be derived from our axiomatization.

Theorem 4.6 ***(Equivalence with FTA axiomatization)***
The following judgments can be derived from the axiomatization 4.1:

(1). ap_irr : $\forall x.\ \neg(x \# x)$
(2). ap_sym : $\forall x, y.\ (x \# y) \Rightarrow (y \# x)$
(3). ap_cot : $\forall x, y.\ (x \# y) \Rightarrow \forall z.\ (x \# z) \vee (z \# y)$
(4). ap_tight : $\forall x, y.\ \neg(x \# y) \Leftrightarrow (x = y)$
(5). add_strext : $\forall x, y, z, u.\ (x + y) \# (z + u) \Rightarrow (x \# z) \vee (y \# u)$
(6). minus_strext : $\forall x, y.\ (-x) \# (-y) \Rightarrow (x \# y)$
(7). mult_strext : $\forall x, y, z, u.\ (x * y) \# (z * u) \Rightarrow (x \# z) \vee (y \# u)$
(8). non_triv : $1 \# 0$
(9). rcpcl_ap_zero : $\forall x.\ (x \# 0) \Rightarrow (x^{-1} \# 0)$
(10). rcpcl_strext : $\forall x, y.\ (x^{-1}) \# (y^{-1}) \Rightarrow (x \# y)$
(11). less_strext : $\forall x, y, z, u.\ (x < y) \Rightarrow (z < u) \vee (x \# z) \vee (y \# u)$
(12). less_trans : $\forall x, y, z.\ (x < y) \wedge (y < z) \Rightarrow (x < z)$
(13). less_irr : $\forall x.\ \neg(x < x)$
(14). add_resp_less : $\forall x, y, z.\ (x < y) \Rightarrow (x + z < y + z)$
(15). times_resp_pos : $\forall x, y.\ (0 < x) \wedge (0 < y) \Rightarrow (0 < x * y)$
(16). less_conf_ap : $\forall x, y, z.\ (x \# y) \Leftrightarrow (x < y) \vee (y < x)$
(17). lim_proof : $\forall s : \mathbb{N} \to F,\ \forall c : \texttt{cauchy}(s).\ \forall \epsilon \in F.\ (0 < \epsilon) \Rightarrow \exists n \in \mathbb{N}.\ \forall m \in \mathbb{N}.\ (n \leq m) \Rightarrow |\, s(m) - \texttt{lim}(s, c)\,| < \epsilon$

Proof. (1). (Irreflexivity of apartness): by point (13).

(2). (Symmetry): by definition.

(3). (Co-transitivity): by the $<$-*co-transitivity* axiom.

(4). (Tightness): by definition.

(5). (Strong extensionality of addition): by lemma 4.12.(1).

(6). (Strong extensionality of the negation function): by points (2), (5) and lemma 4.11.(5).

(7). (Strong extensionality of multiplication): by lemma 4.13.(1).

(8). (Non triviality): by our *non triviality* axiom.

(9). (Reciprocal respects apartness from zero): starting from $x \# 0$ and the non triviality $1 \# 0$ we have both $(x * (x^{-1})) = 1$ by the $\times$-*unit* axiom and $(x * 0) = 0$ by lemma 4.13.(4). Next we derive $(x * x^{-1}) \# (x * 0)$ by lemma 4.11.(5), thus concluding through the strong extensionality of the multiplication.

(10). (Strong extensionality of the reciprocal function): since $x^{-1} = x^{-1} * (y * y^{-1})$ and $y^{-1} = y^{-1} * (x * x^{-1})$, we deduce $(x^{-1} * y) * y^{-1} \# (x^{-1} * x) * y^{-1}$ by lemma 4.11.(5) and $\times$-*associativity*. Using the strong extensionality of the multiplication, we have first $(x^{-1} * y) \# (x^{-1} * x)$ and also $(y \# x)$, so we conclude by point (2).

(11). (Strong extensionality of order): by $<$-*co-transitivity*.

(12). (Transitivity): by lemma 4.11.(2).

(13). (Irreflexivity): by lemma 4.11.(1).

(14). (Addition respects order): by lemma 4.12.(3).

(15). (Multiplication respects positivity): by lemma 4.13.(6).

(16). (Order confirms apartness): this coincides with our definition of apartness.

(17). This axiom is at least as strong as the one we have stated in our axiomatization. We can deduce the FTA axiom from ours as follows: in order to calculate the limit of a generic Cauchy sequence S we need to be able to extract a subsequence of S converging with an exponential convergence rate. This subsequence can be obtained once we know the convergence rate of S, which in turn can be extracted, using the Axiom of Choice, from the proposition stating that S satisfies the Cauchy condition. □

Bridges' axiomatization.

Bridges [Bri99] uses the framework of Bishop's constructive mathematics [Bis67] for presenting an axiomatization of the real line. His main motivation coincides only partially with ours: the curiosity of the mathematician about the properties that suffice to characterize the real numbers and to develop the real analysis. The perspective of the author is to capture the traditional idea that a real could be approximated by arbitrarily close rational numbers.

The constructive axiomatization given by Bridges collects 20 axioms: the properties are quite similar to ours apart the axioms for the order relation and the completeness one.

The structure. The constructive reals are assumed as a set $\mathbf{R}$ together with the neuter elements 0 and 1, the binary relation $>$, the binary operations $+$ and $\cdot$, the unary operations $-$ and $^{-1}$.

The extra relations $\neq$ and $=$ are defined via the order, respectively by:

$$\begin{array}{lcl} (x \neq y) & \triangleq & (x > y) \vee (y > x) \\ (x = y) & \triangleq & (x \geq y) \wedge (y \geq x) \\ (x \geq y) & \triangleq & (\forall z.\ (y > z) \Rightarrow (x > z)) \end{array}$$

Using the points (4), (5) of lemma 4.14 we have that $\forall z.\ (y > z) \Rightarrow (x > z)$ if and only if $\neg(x < y)$. The same could be proved using Bridges' axioms. Therefore Bridges' approach is equivalent to ours up to the use of the more involved definition of the equivalence relation "=".

Field. The axioms concerning the field structure coincide with ours; according to Bridges' terminology, they give rise to an *Heyting field*:

$$\begin{array}{l} x + y = y + x \\ (x + y) + z = x + (y + z) \\ 0 + x = x \\ x + (-x) = 0 \\ xy = yx \\ (xy)z = x(yz) \\ 1 * x = x \\ xx^{-1} = 1 \text{ (if } x \neq 0) \\ x(y + z) = xy + xz \end{array}$$

Bridges assumes as extra implicit axioms the *extensionality* of the relations and operations, i.e. they preserve the equivalence. As already remarked, these are just derived properties in our approach.

Ordered field. The axioms about the order are the following:

$$\neg(x > y \wedge y > x)$$
$$(x > y) \Rightarrow \forall z.\ (x > z) \vee (z > y)$$
$$\neg(x \neq y) \Rightarrow (x = y)$$
$$(x > y) \Rightarrow \forall z.\ (x + z > y + z)$$
$$(x > 0 \wedge y > 0) \Rightarrow (xy > 0)$$

The main difference with respect to our axiomatization is that Bridges requires that the operations *preserve* the order, whereas we require the operations *reflect* it.

The first axiom is equivalent to our $<$-*asymmetry*; the second one coincides with $<$-*co-transitivity*; the third one is trivially valid in our approach, because of our direct representation of the two notions via the strict order —in particular we have also the converse $(x = y) \Rightarrow \neg(x \neq y)$. The last two axioms have been derived in lemmas 4.12.(3) and 4.13.(6).

Archimedeanity. Let be $\mathbf{Z}$ the set of the integer numbers; the axiom used by Bridges is equivalent to ours:

$$\forall x \in \mathbf{R}.\ \exists n \in \mathbf{Z}.\ (x < n)$$

Completeness. This axiom is quite different from ours: the completeness of the real line is postulated through a *least-upper-bound principle*, requiring that every *strongly bounded* set of reals has a least upper bound (l.u.b.). Bridges deduces from this principle that every Cauchy sequence has limit.

In order to justify his axiom, Bridges shows in [BR99] that the existence of the l.u.b. of strongly bounded sets can be derived from the existence of the limit for Cauchy sequences. Hence this approach to the completeness is equivalent to ours. We prefer to state the completeness in terms of Cauchy sequences, because it is simpler.

We can conclude that the different and extra axioms used by Bridges can be derived from our axiomatization.

Theorem 4.7 ***(Equivalence with Bridges' axiomatization)***
The axioms used by Bridges can be derived from the axiomatization 4.1.

Proof. By the above comparison. □

4.5 Completeness and categoricity

In this chapter we have proposed an axiomatization of the constructive real numbers and we have focused on the only two alternative ones in the literature. As far as we know, Bridges and the FTA group have elaborated their axioms independently; in parallel, we have stated our axioms without being aware of the work by Bridges.

The three axiomatizations have the same deductive power. We claim that ours has the advantage to be simpler and more compact. In particular we avoid the redundancy of the FTA axiomatization and we give a more direct treatment of the equivalence relation ($\sim$) than Bridges; we use the reflection of addition and multiplication, which is more powerful

than the preservation; finally our completeness axiom is simpler than the corresponding one in any of the other proposals.

It is natural to raise the fundamental question about the *completeness* of the axiomatization 4.1. The equivalence with the two alternative contributes in the literature seems a positive argument. In particular, the work [GN01] gives also a proof in Coq that the FTA axiomatization is *categorical*, i.e. that any two models are isomorphic. Since our axiomatization is equivalent to the FTA one, we can deduce that our axiomatization is categorical as well.

4.6 Conclusion

We have carried out a construction of the real numbers in Coq using co-recursive streams and constructive logic. Then we have proved that it is adequate w.r.t. a second order axiomatization, which we have proposed and motivated. Hence our streams can be used as a concrete implementation for computing and for addressing formally the reliability of exact algorithms on reals. This fact is very important from the point of view of software engineering, because it allows for the development of certified programs working on the reals. This feature is advocated by constructive mathematicians and computer scientists interested in the development of dependable information technology, as Martin-Löf [NPS90], who has particularly appreciated an early presentation of our work[3].

Our effort has been fruitful also in order to devise a suitable characterization of the constructive reals: we have synthesized a minimal axiomatization and we have proved the equivalence with the alternative ones of the literature.

In the investigation we have mainly used co-inductive tools. Our formal development shows the importance of the co-inductive principles in theoretical computer science and shows that they are the more natural and powerful ones for dealing with circular, non well-founded entities. We see a lacking of these technologies in the current generation of proof assistants and theorem provers. Coq is the only proof assistant embarking a proof tactic specific for dealing with co-inductive assertions: its pragmatic is extremely successful in the present case, but, more generally, it is still hard working with co-inductive definitions and judgments in Coq in a "guarded" way.

Finally, this whole case-study witnesses that the Coq environment is a suitable system for developing axiomatic and model-theoretic approaches to the certification of software.

Future work. We are interested to achieve in the future the following goals:

- to design and implement more advanced exact algorithms working on co-recursive streams, starting from the standard analytic functions *sin*, *cos*, *exp* and *log* [BC90];
- to extract and use exact algorithms in lazy functional programming languages (e.g. `Haskell`);
- to construct other, eventually more efficient, models of the constructive real numbers;
- to look for scientific exchange and cooperation with research programs similar to ours.

[3] Types Annual Meeting, Durham (UK), 08-12 December 2000.

Concerning the axiomatic approach, a possible direction for future work is to consider an axiomatization for the constructive reals not requiring the Axiom of Choice. In this perspective, it would be interesting to consider also a constructive axiomatization obtained by Dedekind cuts: Cauchy sequences and Dedekind cuts provide actually equivalent constructions for the reals only in the case the Axiom of Choice is available [TvD88]. Results in this sense would help to characterize the fundamental differences between the two constructions.

Part III

Objects

Chapter 5

Abadi and Cardelli's impς-calculus

In this chapter we introduce and survey the **imp**ς-calculus of Abadi and Cardelli [AC96]: a pure, imperative, untyped calculus of objects forming the kernel of the Obliq [Car95] programming language.

5.1 Overview of the calculus

The **imp**ς-calculus is a tiny but expressive object-based untyped imperative calculus, which comprises objects, method invocation, method update, object cloning and local definitions:

a, b	$::=$	x	variable
		$[l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}]$	object (l_i distinct)
		$a.l$	method invocation
		$a.l \leftarrow \varsigma(x)b$	method update
		$clone(a)$	cloning
		$let\ x = a\ in\ b$	let

An *object* $[l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}]$ is a collection of components $l_i = \varsigma(x_i)b_i$ for distinct method names l_i and associated methods $\varsigma(x_i)b_i$. The letter ς (sigma) is used as binder for the *self* parameter of a method: $\varsigma(x)b$ is a method with self parameter x and body b. The parameter x has to be bound to the method's *host* object, the object containing the given method. Since the binders $\varsigma(x_i)$ suspend the evaluation of the bodies b_i, the order of the components does not matter. The calculus **imp**ς dispenses with the fields, because the method suite is mutable; fields could be eventually defined using the *let* construct.

Method invocation $a.l$, where the method named l in a is $\varsigma(x)b$, has the intent of executing the body b with the parameter x bound to the host object a, thus returning the result of the execution.

Method update $a.l \leftarrow \varsigma(x)b$ is a typical object-based imperative operation: it replaces the method named l in a with $\varsigma(x)b$ and then returns the modified object.

The *cloning* operation $clone(a)$ is characteristic of prototype-based languages: it produces a new object with the same labels of a, with each component sharing the methods of the corresponding component of a.

The *let* construct $let\ x = a\ in\ b$ evaluates a term a, binds the result to a variable x and then evaluates a second term b with the variable x in the scope, thus permitting to have

local definitions and to control the execution flow. For example, it is possible to define sequential evaluation as follows:

$$a; b \triangleq \mathit{let}\ x = a\ \mathit{in}\ b \qquad \text{if } x \notin FV(b)$$

The semantics of **impς** is given in a natural semantics style *à la* Kahn [Kah87].

Operational semantics. The operational semantics is a big-step one. It is expressed by a reduction relation relating a store σ, a stack S, a term a, a result v and another store σ':

$$\sigma \cdot S \vdash a \rightsquigarrow v \cdot \sigma'$$

The intended meaning is that, given the store σ and the stack S, the term a reduces to a result v, yielding an updated store σ' and leaving the stack S unchanged in the process.

An intuitive explanation of the entities used by the operational semantics is as follows. The global *store* is a function mapping locations to method *closures*. Closures are pairs built of *methods* and *stacks*; stacks are used for the reduction of the corresponding method *bodies*: they associate variables with object *results*. Results are sequences of pairs: method labels together with store locations, one location for each object component. Notice that, in order to stay close to standard implementation techniques, there is no use of formal substitution. We list the entities discussed so far:

ι			store location
v	::=	$[l_i = \iota_i{}^{i\in 1..n}]$	result
S	::=	$x_i \mapsto v_i{}^{i\in 1..n}$	stack
σ	::=	$\iota_i \mapsto \langle \varsigma(x_i) b_i, S_i \rangle^{i\in 1..n}$	store

Stores and stacks are represented by finite sequences: $\iota_i \mapsto c_i{}^{i\in 1..n}$ is the store that maps the location ι_i to the closure c_i, for $i \in 1..n$; $\sigma, \iota \mapsto c$ extends σ; $\sigma.\iota \leftarrow c$ updates σ.

The judgments used in the operational semantics are *well-formedness of stores* $\sigma \vdash \diamond$, *well-formedness of stacks* $\sigma \cdot S \vdash \diamond$ and *term reduction* $\sigma \cdot S \vdash a \rightsquigarrow v \cdot \sigma'$:

$$(\text{Store } \emptyset)\frac{}{\emptyset \vdash \diamond} \qquad (\text{Store } \iota)\frac{\sigma \cdot S \vdash \diamond \quad \iota \notin \mathit{dom}(\sigma)}{\sigma, \iota \mapsto \langle \varsigma(x) b, S \rangle \vdash \diamond} \qquad (\text{Stack } \emptyset)\frac{\sigma \vdash \diamond}{\sigma \cdot \emptyset \vdash \diamond}$$

$$(\text{Stack } x)\frac{\sigma \cdot S \vdash \diamond \quad \iota_i \in \mathit{dom}(\sigma) \quad x \notin \mathit{dom}(S) \quad (l_i, \iota_i\ \mathit{distinct}) \quad \forall i \in 1..n}{\sigma \cdot (S, x \mapsto [l_i = \iota_i{}^{i\in 1..n}]) \vdash \diamond}$$

A well-formed store is built starting from the well-formed empty sequence and allocating a new closure pointed to by a fresh pointer, provided the closure contains a well-formed stack. The empty well-formed stack is built starting from a well-formed store; a well-formed stack can be enriched pushing on its top the association between a fresh variable and the result formed by pairs of distinct labels and distinct, not dangling, pointers to the store.

The main term reduction judgment, which relates terms to results in stores and forms

the core of the operational semantics, is defined by the following structural rules:

$$\text{(Red } x\text{)}\frac{\sigma \cdot (S', x \mapsto v, S'') \vdash \diamond}{\sigma \cdot (S', x \mapsto v, S'') \vdash x \rightsquigarrow v \cdot \sigma}$$

$$\text{(Red Object)}\frac{\sigma \cdot S \vdash \diamond \quad \iota_i \notin dom(\sigma) \quad (l_i, \iota_i \; distinct) \quad \forall i \in 1..n}{\sigma \cdot S \vdash [l_i = \varsigma(x_i)b_i{}^{i\in 1..n}] \rightsquigarrow [l_i = \iota_i{}^{i\in 1..n}] \cdot (\sigma, \iota_i \mapsto \langle \varsigma(x_i)b_i, S\rangle^{i\in 1..n})}$$

$$\text{(Red Select)}\frac{\begin{array}{c}\sigma \cdot S \vdash a \rightsquigarrow [l_i = \iota_i{}^{i\in 1..n}] \cdot \sigma' \quad \sigma'(\iota_j) = \langle \varsigma(x_j)b_j, S'\rangle \quad j \in 1..n \\ x_j \notin dom(S') \qquad \sigma' \cdot (S', x_j \mapsto [l_i = \iota_i{}^{i\in 1..n}]) \vdash b_j \rightsquigarrow v \cdot \sigma''\end{array}}{\sigma \cdot S \vdash a.l_j \rightsquigarrow v \cdot \sigma''}$$

$$\text{(Red Update)}\frac{\sigma \cdot S \vdash a \rightsquigarrow [l_i = \iota_i{}^{i\in 1..n}] \cdot \sigma' \quad j \in 1..n \quad \iota_j \in dom(\sigma')}{\sigma \cdot S \vdash a.l_j \leftarrow \varsigma(x)b \rightsquigarrow [l_i = \iota_i{}^{i\in 1..n}] \cdot (\sigma'.\iota_j \leftarrow \langle \varsigma(x)b, S\rangle)}$$

$$\text{(Red Clone)}\frac{\begin{array}{c}\iota_i \in dom(\sigma') \quad \iota_i{}' \notin dom(\sigma') \quad (\iota_i{}' \; distinct) \quad \forall i \in 1..n \\ \sigma \cdot S \vdash a \rightsquigarrow [l_i = \iota_i{}^{i\in 1..n}] \cdot \sigma'\end{array}}{\sigma \cdot S \vdash clone(a) \rightsquigarrow [l_i = \iota_i{}'{}^{i\in 1..n}] \cdot (\sigma', \iota_i{}' \mapsto \sigma'(\iota_i)^{i\in 1..n})}$$

$$\text{(Red Let)}\frac{\sigma \cdot S \vdash a \rightsquigarrow v' \cdot \sigma' \quad \sigma' \cdot (S, x \mapsto v') \vdash b \rightsquigarrow v'' \cdot \sigma''}{\sigma \cdot S \vdash let \; x = a \; in \; b \rightsquigarrow v'' \cdot \sigma''}$$

A *variable* reduces to the result it denotes in the current stack. An *object* reduces to a result consisting of a fresh collection of locations, such that the store is extended for associating the method closures to these locations. A *selection* operation first reduces its host object to a result, then activates the appropriate method closure. An *update* operation reduces its object and then updates the appropriate store location with a new method closure. A *cloning* operation reduces its object, then allocates a fresh collection of locations and associates them to the existing method closures from the object (deep cloning). A *let* construct reduces to the result of reducing its body in a stack extended with the bound variable associated to the result of its local term.

Finally, notice that an algorithm for reduction can be extracted from the rules: it would parallel standard implementations of objects.

Type system. The type system for the untyped **imp**ς-calculus is a first-order one with subtyping. The only type constructor is the one for object types:

$$A, B \quad ::= \quad [l_i : B_i{}^{i\in 1..n}] \qquad \text{object type } (l_i \text{ distinct})$$

Notice that the only ground type is $[\,]$, which can be used as a starting point for building object types; other ground types, as $Bool$ or Nat, can can be added at will.

The formal typing system is given by four judgments, whose general form is $E \vdash \mathcal{I}$, where E is a typing environment consisting of a list of assumptions for variables, each of the form $x : A$; the shape of $\mathcal{I}$ depends on the judgment.

The first judgment $E \vdash \diamond$ describes how to build *well-formed type environments*:

$$\text{(Env } \emptyset\text{)}\frac{}{\emptyset \vdash \diamond} \qquad \text{(Env x)}\frac{E \vdash A \quad x \notin dom(E)}{E, x : A \vdash \diamond}$$

The empty environment $\emptyset$ is well-formed; a new association $x{:}A$ extends a well-formed environment, provided x is a fresh variable and A a well-formed type.

The second judgment $E \vdash B$ states that B is a *well-formed type* in the environment E; the unique rule concerns the formation of object types:

$$\text{(Type Object)}\frac{E \vdash B_i \quad (l_i \ distinct) \quad \forall i \in 1..n}{E \vdash [l_i : B_i{}^{i\in 1..n}]}$$

An object type $[l_i : B_i{}^{i\in 1..n}]$ is well-formed in the environment E provided that each B_i is well formed in E and that the labels l_i are distinct.

The third judgment introduces the notion of *subsumption*, which is induced by a *subtype* relation $A <: B$ between object types. An object belonging to a given object type belongs to any supertype of that type as well, and can subsume objects in the supertype, because these have a more limited protocol:

$$\text{(Sub Refl)}\frac{E \vdash A}{E \vdash A <: A}$$

$$\text{(Sub Trans)}\frac{E \vdash A <: B \quad E \vdash B <: C}{E \vdash A <: C}$$

$$\text{(Sub Object)}\frac{E \vdash B_i \quad (l_i \ distinct) \quad \forall i \in 1..n+m}{E \vdash [l_i : B_i{}^{i\in 1..n+m}] <: [l_i : B_i{}^{i\in 1..n}]}$$

The first two rules are the basic reflexivity and transitivity; the third one allows a longer object type to be a subtype of a shorter one. Notice that object types are *invariant* in their component types: the subtyping $[l_i : B_i{}^{i\in 1..n+m}] <: [l_i : C_i{}^{i\in 1..n}]$ requires $B_i \equiv C_i$ for all $i \in 1..n$; that is, object types are neither covariant neither contravariant in their component types. This condition is necessary in order to guarantee the soundness of the type discipline.

Finally, the main *term typing judgment* $E \vdash a : A$ states that a has type A in E:

$$\text{(Val Subsumption)}\frac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B}$$

$$\text{(Val x)}\frac{E', x : A, E'' \vdash \diamond}{E', x : A, E'' \vdash x : A}$$

$$\text{(Val Object)}\frac{E, x_i : [l_i : B_i{}^{i\in 1..n}] \vdash b_i : B_i \quad \forall i \in 1..n}{E \vdash [l_i : \varsigma(x_i)b_i{}^{i\in 1..n}] : [l_i : B_i{}^{i\in 1..n}]}$$

$$\text{(Val Select)}\frac{E \vdash a : [l_i : B_i{}^{i\in 1..n}] \quad j \in 1..n}{E \vdash a.l_j : B_j}$$

$$\text{(Val Update)}\frac{E \vdash a : [l_i : B_i{}^{i\in 1..n}] \quad E, x : [l_i : B_i{}^{i\in 1..n}] \vdash b : B_j \quad j \in 1..n}{E \vdash a.l_j \leftarrow \varsigma(x)b : [l_i : B_i{}^{i\in 1..n}]}$$

$$\text{(Val Clone)}\frac{E \vdash a : [l_i : B_i{}^{i\in 1..n}]}{E \vdash clone(a) : [l_i : B_i{}^{i\in 1..n}]}$$

$$\text{(Val Let)}\frac{E \vdash a : A \quad E, x : A \vdash b : B}{E \vdash let\ x = a\ in\ b : B}$$

The rule ($Val\ Subsumption$) connects the typing to the subtyping: an object can emulate another object that has fewer methods, i.e. a more limited protocol. ($Val\ x$) is used to extract an assumption from an environment, where x occurs somewhere in. According to ($Val\ Object$), an object type $[l_i : B_i{}^{i\in 1..n}]$ can be assigned to a collection of n methods whose bodies have types $B_1, \ldots, B_n$. (Notice the circularity introduced by the *self* parameter: in order to give a value a type $[l_i : B_i{}^{i\in 1..n}]$ the existence of a value of the same type is assumed.) The rule ($Val\ Select$) tells that, when a method l_j of an object type $[l_i : B_i{}^{i\in 1..n}]$ is invoked, it produces the corresponding result type B_j. Method update ($Val\ Update$) preserves the type of the object that is updated: the type of the object cannot be allowed to change, because other methods assume it. (An update may modify an object whose complete set of methods is statically unknown.) The last two rules ($Val\ Clone$) and ($Val\ Let$) do not require any specific explanation.

5.2 Examples

The **imp**ς-calculus is tiny and economical, but fairly expressive and complete, as we point out in the current section.

Notice first that it is possible to translate the **imp**ς-calculus into its *functional* version (the ς-calculus [AC96], chapter 6) just expressing the functional method update as cloning followed by imperative method update.

An equivalent alternative to the **imp**ς-calculus is the direct availability of fields, field selection and field update in the syntax. This is obtained with **imp**ς_f, a calculus with eagerly evaluated fields, such that the semantics of the objects depends on the order of their components:

a, b	::=	x	variable
		$[l_i = b_i{}^{i\in 1..n},\ l_j = \varsigma(x_j)b_j{}^{j\in n+1..n+m}]$	object (l_i, l_j distinct)
		$a.l$	field selection/method invocation
		$a.l := b$	field update
		$a.l \leftarrow \varsigma(x)b$	method update
		$clone(a)$	cloning

The **imp**ς and **imp**ς_f calculi are inter-translatable.

A further possibility for the imperative objects of **imp**ς is to express procedures. This feature is captured by the **imp**λ-calculus, a call-by-value λ-calculus with side-effects, including abstraction, application and assignment to λ-bound variables:

a, b	::=	x	variable
		$x := a$	assignment
		$\lambda(x)b$	abstraction
		$b(a)$	application

The **imp**λ-calculus can be extended into **imp**$\lambda\varsigma_f$, a calculus with procedures and objects, equipped with procedures with call-by-value parameters and assignable formals, and objects with fields and methods. Its syntax is simply the union of those ones of **imp**λ and **imp**ς_f. Also this calculus is inter-translatable with **imp**ς.

It would be convenient to adopt the **imp**$\lambda\varsigma_f$-calculus for a synthetic development of complex examples: for instance, it allows to encode basic types (as booleans, and natural numbers) and iteration constructs. Anyway, for coherence with our work in following chapters, we prefer, from now on, to focus on the primary **imp**ς-calculus. We detail now reduction and typing examples for illustrating the behavior of the imperative terms and some peculiarities of **imp**ς.

Reduction. The examples we are discussing are the following:

1. $let\ y = [l = \varsigma(x)x]\ in\ clone(y)$
2. $[l = \varsigma(x)x.l].l$
3. $[l = \varsigma(x)x.l \leftarrow \varsigma(y)x].l$

The reduction of the first term shows how to handle variables, objects, local declarations and cloning. First we reduce the object $[l = \varsigma(x)x]$:

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \diamond}\ (Store\ \emptyset)}{\emptyset \cdot \emptyset \vdash \diamond}\ (Stack\ \emptyset)}{\emptyset \cdot \emptyset \vdash [l = \varsigma(x)x] \rightsquigarrow [l = 0] \cdot \{0 \mapsto \langle \varsigma(x)x, \emptyset \rangle\}}\ (Red\ Object)$$

Then we consider the body of the *let* construct, denoting σ_1 the store $\{0 \mapsto \langle \varsigma(x)x, \emptyset \rangle\}$, obtained above:

$$\dfrac{\dfrac{\dfrac{\cdots}{\sigma_1 \cdot (y \mapsto [l = 0]) \vdash \diamond}\ (Stack\ x)}{\sigma_1 \cdot (y \mapsto [l = 0]) \vdash y \rightsquigarrow [l = 0] \cdot \sigma_1}\ (Red\ x)}{\sigma_1 \cdot (y \mapsto [l = 0]) \vdash clone(y) \rightsquigarrow [l = 1] \cdot \{\sigma_1, 1 \mapsto \langle \varsigma(x)x, \emptyset \rangle\}}\ (Red\ Clone)$$

Finally we apply the (*Red Let*) rule, thus concluding:

$$\emptyset \cdot \emptyset \vdash let\ y = [l = \varsigma(x)x]\ in\ clone(y) \rightsquigarrow [l = 1] \cdot \{0 \mapsto \langle \varsigma(x)x, \emptyset \rangle, 1 \mapsto \langle \varsigma(x)x, \emptyset \rangle\}$$

The second example, which illustrates method selection, gives rise to a divergent reduction. The preliminary step is the following:

$$\dfrac{\cdots}{\emptyset \cdot \emptyset \vdash [l = \varsigma(x)x.l] \rightsquigarrow [l = 0] \cdot \{0 \mapsto \langle \varsigma(x)x.l, \emptyset \rangle\}}\ (Red\ Object)$$

Let us denote σ_2 the store $\{0 \mapsto \langle \varsigma(x)x.l, \emptyset \rangle\}$; then:

$$\dfrac{\dfrac{\cdots}{\sigma_2 \cdot (x \mapsto [l = 0]) \vdash \diamond}\ (Stack\ x)}{\sigma_2 \cdot (x \mapsto [l = 0]) \vdash x \rightsquigarrow [l = 0] \cdot \sigma_2}\ (Red\ x)$$

Let us denote $J_{Red(x)}$ the judgment $\sigma_2 \cdot (x \mapsto [l = 0]) \vdash x \rightsquigarrow [l = 0] \cdot \sigma_2$ obtained above. The divergent reduction results from a repeating pattern on an infinite branch, which allocates infinitely variables on the stack:

$$\dfrac{J_{Red(x)} \qquad \dfrac{J_{Red(x,y)} \qquad \cdots}{\sigma_2 \cdot (x, y \mapsto [l = 0]) \vdash y.l \rightsquigarrow ? \cdot ?}\ (Red\ Select)}{\sigma_2 \cdot (x \mapsto [l = 0]) \vdash x.l \rightsquigarrow ? \cdot ?}\ (Red\ Select)$$

Thus also the main term diverges:

$$\dfrac{\begin{array}{c}\sigma_2 \cdot (x \mapsto [l = 0]) \vdash x.l \rightsquigarrow ? \cdot ? \\ \emptyset \cdot \emptyset \vdash [l = \varsigma(x)x.l] \rightsquigarrow [l = 0] \cdot (0 \mapsto \langle \varsigma(x)x.l, \emptyset \rangle)\end{array}}{\emptyset \cdot \emptyset \vdash [l = \varsigma(x)x.l].l \rightsquigarrow ? \cdot ?}\ (Red\ Select)$$

The final example concerns method update and points out the critical problem of having loops in the store:

$$\dfrac{\cdots}{\emptyset \cdot \emptyset \vdash [l = \varsigma(x)x.l \leftarrow \varsigma(y)x] \rightsquigarrow [l = 0] \cdot \{0 \mapsto \langle \varsigma(x)x.l \leftarrow \varsigma(y)x, \emptyset \rangle\}}\ (Red\ Object)$$

Let us denote σ_3 the store $\{0 \mapsto \langle \varsigma(x)x.l \leftarrow \varsigma(y)x, \emptyset \rangle\}$; therefore we have:

$$\dfrac{\dfrac{\dfrac{\cdots}{\sigma_3 \cdot (x \mapsto [l = 0]) \vdash \diamond}\ (Stack\ x)}{\sigma_3 \cdot (x \mapsto [l = 0]) \vdash x \rightsquigarrow [l = 0] \cdot \sigma_3}\ (Red\ x)}{\sigma_3 \cdot (x \mapsto [l = 0]) \vdash x.l \leftarrow \varsigma(y)x \rightsquigarrow [l = 0] \cdot \{0 \mapsto \langle \varsigma(y)x, (x \mapsto [l = 0]) \rangle\}}\ (Red\ Update)$$

Hence we can apply the (*Red Select*) rule, thus concluding:

$$\emptyset \cdot \emptyset \vdash [l = \varsigma(x)x.l \leftarrow \varsigma(y)x].l \rightsquigarrow [l = 0] \cdot \{0 \mapsto \langle \varsigma(y)x, (x \mapsto [l = 0]) \rangle\}$$

The resulting store $\{0 \mapsto \langle \varsigma(y)x,\ (x \mapsto [l = 0]) \rangle\}$ contains a loop, because it maps the address 0 to a closure that binds the variable x to a result that contains again the location 0. The potential presence of loops in the store, which is characteristic of imperative semantics, complicates the reasoning about programs and demands special attention in the treatment of the Type Soundness.

Typing. We illustrate the relative simplicity of the imperative typing: a method may store results in global locations, but its result type does not reflect this fact. We reconsider the first two reduction examples above.

The typing of term *let* $y = [l = \varsigma(x)x]$ *in clone*(y) requires first to type $[l = \varsigma(x)x]$. Since $\emptyset \vdash [\,]$, we have immediately that:

$$\dfrac{\emptyset \vdash [\,]}{\emptyset \vdash [l : [\,]]}\ (Type\ Object) \qquad\qquad \dfrac{\emptyset \vdash [\,]}{\emptyset \vdash [l : [\,]] <: [\,]}\ (Sub\ Object)$$

Thus we can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\emptyset \vdash [l : [\,]]}{x : [l : [\,]] \vdash \diamond}\ (Env\ x)}{x : [l : [\,]] \vdash x : [l : [\,]]}\ (Val\ x) \qquad \emptyset \vdash [l : [\,]] <: [\,]}{x : [l : [\,]] \vdash x : [\,]}\ (Val\ Subsumption)}{\emptyset \vdash [l = \varsigma(x)x] : [l : [\,]]}\ (Val\ Object)$$

On the other hand, we have:

$$\dfrac{\dfrac{y : [l : [\,]] \vdash \diamond}{y : [l : [\,]] \vdash y : [l : [\,]]}\ (Val\ x)}{y : [l : [\,]] \vdash clone(y) : [l : [\,]]}\ (Val\ Clone)$$

Hence we can apply the $(Val\ Let)$ rule, thus verifying that the term can be given its minimum type:

$$\emptyset \vdash let\ y = [l = \varsigma(x)x]\ in\ clone(y) : [l : [\,]]$$

The second example witnesses that there exist diverging terms which can be typed:

$$\dfrac{\dfrac{\dfrac{\dfrac{x : [l : [\,]] \vdash \diamond}{x : [l : [\,]] \vdash x : [l : [\,]]}\ (Val\ x)}{x : [l : [\,]] \vdash x.l : [\,]}\ (Val\ Select)}{\emptyset \vdash [l = \varsigma(x)x.l] : [l : [\,]]}\ (Val\ Object)}{\emptyset \vdash [l = \varsigma(x)x.l].l : [\,]}\ (Val\ Select)$$

5.3 Type Soundness

The type system of **impς** captures statically the `message-not-found` type error, and the type checking is decidable.

Result typing. The proof of the Type Soundness relies on the validity of the Subject Reduction property, stating that the dynamic semantics is consistent with the static semantics. In turn, the proof of the subject reduction requires to be able to give types to results; indeed, the typing of results is delicate, because results are pointers to the store, and stores may contain loops. Hence is not possible to determine the type of a result examining its substructures recursively. We present below the notions that Abadi and Cardelli introduce for typing results and proving the subject reduction.

The *store types* allow to type results independently of particular stores: this is possible because type-sound computations do not store results of different types in the same location. A store type associates a *method type* to each store location. Method types have

the form $[l_i : B_i{}^{i\in 1..n}] \Rightarrow B_j$, where $[l_i : B_i{}^{i\in 1..n}]$ is the type of *self* and B_j, such that $j \in 1..n$, is the result type:

$$\begin{array}{lcll} \Sigma & ::= & \iota_i \mapsto M_i{}^{i\in 1..n} & \text{store type } (\iota_i \text{ distinct}) \\ M & ::= & [l_i : B_i{}^{i\in 1..n}] \Rightarrow B_j & \text{method type } (j \in 1..n) \\ \Sigma_1(\iota) & \triangleq & [l_i : B_i{}^{i\in 1..n}] & \text{if } \Sigma(\iota) \equiv [l_i : B_i{}^{i\in 1..n}] \Rightarrow B_j \\ \Sigma_2(\iota) & \triangleq & B_j & \text{if } \Sigma(\iota) \equiv [l_i : B_i{}^{i\in 1..n}] \Rightarrow B_j \end{array}$$

The *well-formedness of method types* $\models M \in Meth$ and *well-formedness of store types* $\Sigma \models \diamond$ judgments are introduced next:

$$\text{(Method Type)}\frac{j \in 1..n}{\models [l_i : B_i{}^{i\in 1..n}] \Rightarrow B_j \in Meth}$$

$$\text{(Store Type)}\frac{\models M_i \in Meth \qquad (\iota_i \; distinct) \qquad \forall i \in 1..n}{\iota_i \mapsto M_i{}^{i\in 1..n} \models \diamond}$$

The subject reduction theorem is stated through the *result typing* judgment $\Sigma \models v : A$, whose intended meaning is that the result v is given the type A, copied from store type Σ:

$$\text{(Result Object)}\frac{\Sigma \models \diamond \qquad \Sigma_1(\iota_i) \equiv [l_i : \Sigma_2(\iota_i)^{\,i\in 1..n}] \qquad \forall i \in 1..n}{\Sigma \models [l_i = \iota_i{}^{i\in 1..n}] : [l_i : \Sigma_2(\iota_i)^{\,i\in 1..n}]}$$

Since results are interpreted in stores, it is necessary to capture the compatibility between stores and store types. This is accomplished by the *store typing* judgment $\Sigma \models \sigma$, whose intended meaning is to check that the content of every store location, i.e. closure, can be given the type of the store type for that location. The store typing judgment allows to type each closure with respect to the whole store, thus accounting for cycles in store:

$$\text{(Store Typing)}\frac{\Sigma \models S_i : E_i \qquad E_i, x_i : \Sigma_1(\iota_i) \vdash b_i : \Sigma_2(\iota_i) \qquad \forall i \in 1..n}{\Sigma \models \iota_i \mapsto \langle \varsigma(x_i)b_i, S_i \rangle^{i\in 1..n}}$$

The method body of a closure is typed using a type environment compatible with the stack contained in that closure. This compatibility is described by the *stack typing* judgment, which is defined via the result typing:

$$\text{(Stack } \emptyset \text{ T.)}\frac{\Sigma \models \diamond}{\Sigma \models \emptyset : \emptyset} \qquad \text{(Stack } x \text{ T.)}\frac{\Sigma \models S : E \qquad \Sigma \models v : A \qquad x \notin dom(E)}{\Sigma \models S, x \mapsto v : E, x{:}A}$$

Example 5.1 *Let us resume the third example of the previous section. We derived:*

$$\emptyset \cdot \emptyset \vdash [l = \varsigma(x)x.l \leftarrow \varsigma(y)x].l \rightsquigarrow [l = 0] \cdot \{0 \mapsto \langle \varsigma(y)x, (x \mapsto [l = 0]) \rangle\} \equiv \sigma$$

We show how to use the judgments introduced in this section for typing the result $[l = 0]$ *with respect to the store* σ*. The key aspect is the role of the store type* Σ*, compatible with the store* σ*; it is necessary to pick out beforehand the right type we want to type-check:*

$$\frac{\Sigma \models \diamond \qquad \Sigma_1(0) \equiv [l : [\,]]}{\Sigma \models [l = 0] : [l : [\,]]} \; (Result\ Object)$$

This means that the store type Σ has form:

$$\{0 \mapsto [l : [\,]] \Rightarrow [\,]\}$$

Hence we are allowed to verify that the store type Σ is compatible with the store σ:

$$\dfrac{\dfrac{\Sigma \models \emptyset : \emptyset \quad \Sigma \models [l = 0] : [l{:}[\,]]}{\Sigma \models x \mapsto [l = 0] : (x{:}[l{:}[\,]])}\ (Stack\ x\ T.) \qquad \dfrac{\cdots}{x : [l{:}[\,]], y : [l{:}[\,]] \vdash x : [\,]}\ (Val\ Sub.)}{\Sigma \models \sigma}\ (Store\ T.)$$

Subject Reduction. The Type Soundness of the typing discipline is a derived ultimate metatheoretical property: it guarantees that the successful static typing of programs is a proof about the partial correctness of their execution.

The Type Soundness for **impς** ensures that every well-typed, not diverging term never yields the `message-not-found` runtime error. This is an immediate consequence of the subject reduction theorem, whose statement and proof requires a preliminary definition and lemma.

Definition 5.1 ***(Store type extension)***
We say that $\Sigma' \geq \Sigma$ (Σ' is an extension of Σ) if and only if:

1. $dom(\Sigma) \subseteq dom(\Sigma')$
2. $\forall \iota \in dom(\Sigma).\ \Sigma'(\iota) = \Sigma(\iota)$

□

Lemma 5.1 ***(Stack typing, Bound weakening)***

1. *If $\Sigma \models S : E$ and $\Sigma' \models \diamond$, with $\Sigma' \geq \Sigma$, then $\Sigma' \models S : E$*
2. *If $E, x : D, E' \vdash \mathcal{I}$ and $E \vdash D' <: D$, then $E, x : D', E' \vdash \mathcal{I}$*

□

The Subject Reduction theorem states that the operational semantics is consistent with the type system.

Theorem 5.1 ***(Subject Reduction)***
If $E \vdash a : A$ and $\sigma \cdot S \vdash a \rightsquigarrow v \cdot \sigma'$ and $\Sigma \models \sigma$ and $dom(\sigma) = dom(\Sigma)$ and $\Sigma \models S : E$, then there exist a type A' and a store type Σ' such that $\Sigma' \geq \Sigma$ and $\Sigma' \models \sigma'$ and $dom(\sigma') = dom(\Sigma')$ and $\Sigma' \models v : A'$ and $A' <: A$.

Proof. *By structural induction on the derivation of $\sigma \cdot S \vdash a \rightsquigarrow v \cdot \sigma'$. See [AC96], page 148.* □

It is immediate to deduce that if a closed term has a type and the term reduces to a result in a store, then the result can be assigned that type in that store. Equivalently, if a closed term produces a result, it does so respecting the type it has been assigned statically.

Corollary 5.1 ***(Subject Reduction for closed terms)***
If $\emptyset \vdash a : A$ *and* $\emptyset \cdot \emptyset \vdash a \rightsquigarrow v \cdot \sigma$*, then there exist a type* A' *and a store type* Σ' *such that* $\Sigma' \models \sigma$ *and* $\Sigma' \models v : A'$*, with* $A' <: A$*.* □

The statement of the previous Corollary is vacuous if the starting term does not produce a result: this can happen either because the reduction diverges —the rules are applicable ad infinitum— or because it gets stuck —no rule is applicable at a certain stage. But the latter case is not possible, which is the essence of the Type Soundness property.

Theorem 5.2 ***(Type Soundness)***
The reduction of a not diverging, well-typed term of $\boldsymbol{imp}_{\subset}$ *in a well-typed store cannot get stuck, and produces a result of the expected type.* □

Chapter 6

Reformulation of impς in Natural Deduction

We reformulate Abadi and Cardelli's **imp**ς-calculus using Natural Deduction and coinductive proof systems; then we prove the Subject Reduction. The new presentation of **imp**ς is better suited for an encoding in Coq, that we carry out in the next chapter.

6.1 Proof-theoretical choices

A reformulation of **imp**ς is a preliminary step taking most advantage of the features provided by the coinductive type theory $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$, underlying Coq. In fact, standard consolidated encoding methodologies of type theory-based logical frameworks (LFs) urge the partial rephrasing of an object system into an equivalent form, better suited w.r.t. this kind of environments. In the present case, we have even the possibility of simplifying the formal proofs about the metatheory of **imp**ς by adopting specific proof-theoretical perspectives supported by the proof assistant Coq. We explain more deeply our motivations as follows.

- LFs allow the user to (re)formulate object systems taking full advantage of proof-theoretical concepts, as for example *Natural Deduction*. In this perspective, LFs may suggest alternative, and cleaner, definitions of the same systems, which are more appropriate for computer-assisted formal reasoning. The operational semantics and typing systems of **imp**ς are expressed in a *Natural Semantics* style à la Kahn [Des86, Kah87], namely using *sequents* logics. However, it turns out that the formalism of the *Natural Operational Semantics* [BH90], a refinement of the original natural semantics approach, is better suited w.r.t. the formalization in a logical framework. This technique permits to give specifications dispensing with the explicit management of structures which obey a stack discipline (as *environments*), thus simplifying both the encoding and the formal proofs. In fact, these structures can be dealt with by means of hypothetical-general premises à la Martin-Löf, that make assumptions about the values of variables.
- The proof of the Subject Reduction for **imp**ς, the core step towards the Type Soundness, is particularly complex and technical already on the paper (as envisaged in the previous chapter). In particular, it requires the introduction of some extra notions

and judgments: this is due essentially to the potential presence of loops in the store, that complicate the formal reasoning. It is well-known that the formalization of a system in a logical framework introduces an implicit overhead, hence we feel useful to state an attempt for simplifying our effort. The idea is to describe circular and infinite entities, as cycles in stores, by means of *coinduction*. So doing, it is possible to avoid the introduction of the extra notions and judgments for proving the Subject Reduction. By the way, the proof assistant Coq supports the use of coinductive definition and proof principles: these ones appear natural and adequate for reasoning about circular and potentially infinite activities, as may be visiting a store with cycles.

This chapter is devoted to make effective the above ideas: we plan to obtain a rephrasing of **imp**ς which is equivalent to the original statement of the calculus and can be easier formalized and worked with in Coq. We spend the remaining part of the section for sketching the techniques and ideas we have picked out in the above discussion.

Natural Semantics with explicit assumptions. Natural operational semantics (NOS) [BH90] is conceived as a technique for treating the operational semantics of languages using the natural deduction style of logics. The main feature of natural deduction proofs, not used in the usual style of semantics —e.g. structural operational semantics [Plo81] and natural semantics [Des86, Kah87]— is that the premises of a rule may be hypothetical: they may be themselves of the form "Q is derivable from P_1 and ... and P_n. This is written:

$$\dfrac{\begin{array}{c}(P_1, \ldots, P_n)\\ \vdots \\ Q\end{array}}{R}$$

Using this setting it is possible to give an operational semantics which dispenses with the common notion of environment. This makes semantic definitions and proofs appreciably simpler than the traditional ones. In fact, the proofs have the same shape of traditional ones, but the environment does not appear repeatedly in the formulas: the role of the environment is played by hypothetical assumptions about the values of the variables which occur in it; that is, only involved *fragments* of the environment are taken into account.

We choose the NOS formalism for defining not only the big-step operational semantics of **imp**ς, but also the term typing and the result typing judgments. Concerning the reduction, we evaluate a term under the hypothetical knowledge about the values associated to its free variables. The same idea is exploited for the other judgments: in those cases the variables are associated to types, as should be in type environments. We call *natural deduction semantics* (NDS) this unified approach for addressing semantics.

The use of the NDS setting is particularly well-suited w.r.t. our goal of reasoning about a calculus in a proof assistant. Having an implicit environment, we are actually freed from all the overhead of managing it, and we can delegate to the metalanguage the allocation of fresh variables that correspond to the involved fragments. (On the other hand, we have to carry out the explicit treatment of the *store*, the other semantic domain required by the semantics, because it cannot be handled implicitly [Mic94]). Another advantage is that we can enjoy the benefits of the natural deduction approach without having to formalize

the *substitution* of terms for variables: this is simply due to the fact that the semantics of **imp**ς does not need substitution.

Coinduction. An alternative to Abadi-Cardelli's approach for typing results —i.e. datatypes containing references to the store— is to adopt coinductive principles. Coinduction reveals to be successful for managing potential cycles in the store, because it deals naturally with circular, non well-founded entities. More precisely, maybe the case that typing a pointer generates the process of visiting store locations, and it is required to type again the pointer itself; in other words, the starting pointer can be reached from the location it refers to. We convey to the reader these intuitive ideas via the motivating example 5.3; let be:

$$\sigma \equiv \{0 \mapsto \langle \varsigma(y)x, (x \mapsto [l = 0])\rangle\}$$

Abadi-Cardelli solve the problem of typing results introducing the structure of *store types*:

$$\frac{\Sigma \equiv 0 \mapsto [l : [\,]] \Rightarrow [\,] \qquad \Sigma \models \diamond \qquad \Sigma_1(0) \equiv [l : [\,]]}{\Sigma \models [l = 0] : [l : [\,]]}\ (\mathit{Result\ Object})$$

Conversely, we advocate a solution which tries to simplify the proof of the Subject Reduction. Our informal idea is to type results following the pointers belonging to the locations pointed to, i.e. in the methods closures. Therefore we continue visiting the store typing closures and pointers, and so on, until a location without pointers is reached: so doing, we avoid to introduce the store types. An informal possible treatment of the above example is the following, where $Type'$ is an auxiliary notion typing the contents of store locations:

$$\dfrac{\dfrac{\dfrac{\vdots}{\sigma \models [l = 0] : [l : [\,]]}\ (\mathit{Result'}) \qquad x, y \mapsto [l : [\,]] \vdash (\mathit{type'}\ \sigma\ \langle x\rangle\ [\,])}{y \mapsto [l : [\,]] \vdash (\mathit{type'}\ \sigma\ \langle x, x \mapsto [l = 0]\rangle\ [\,])}\ (\mathit{Type'})}{\sigma \models [l = 0] : [l : [\,]]}\ (\mathit{Result'})$$

This shows the potential situation of following indefinitely pointers in the store. Hence we decide to model the result typing judgment by coinduction: this perspective is completely new with respect to the original formulation of **imp**ς; we stress that our approach permits to avoid the overhead of introducing the notion of store type and the extra judgments necessary for addressing the Subject Reduction. The price to pay is to devise an appropriate way for typing the results and to certify it.

We remark that, in parallel with our coinductive setting, we develop also Abadi-Cardelli's canonical approach. The comparison between the two formalizations is very illuminating: it shows for example the extra work necessary for adding the extra explicit semantic structure of store types to an already existing formalization; very interesting is also the comparison between the proof techniques induced by the two approaches. This material appears also in [CLM03].

6.2 Dynamic and static semantics

Our rephrasing of **imp**ς derives immediately from the choice of using the NDS approach, which is based on the natural deduction style of proof. The key point is that we do not have environment structures directly in the judgments, because they are distributed in the hypotheses of proof derivations: so doing, we obtain simpler judgments (and proofs), but we have to address the consequences of this choice.

All the judgments we use from now on refer to a context Γ, which is the *proof derivation context*: it comprises an heterogeneous set of hypotheses, which can be used as assumptions in the proof derivations and we present and discuss below. Notice that, when clear from the text, we write $\mathcal{J}$ for the judgment $\Gamma \vdash \mathcal{J}$. As usual in natural deduction, rules are written in "vertical" notation, i.e. the hypotheses of a derivation $\Gamma \vdash \mathcal{J}$ are distributed on the leaves of the proof tree.

The major changes concern the operational semantics, whereas the type system for terms needs only a lighter reformulation. Very delicate is the typing of results, which requires deep proof-theoretical considerations, which we address separately in sections 6.5 and 6.6.

Dynamic semantics.

The main *term reduction* judgment is translated in the following way:

$$\sigma \cdot S \vdash a \rightsquigarrow v \cdot \tau \qquad \longmapsto \qquad \Gamma \vdash (eval\ s\ a\ t\ v)$$

That is, we model term reduction by a predicate *eval* defined on 4-tuples:

$$eval \subseteq Store \times Term \times Store \times Res$$

where $Term$, $Store$, Res correspond directly to the entities used by Abadi-Cardelli. The intended meaning of the derivation of $\Gamma \vdash (eval\ s\ a\ t\ v)$ is that, starting with the store s and using the assumptions in Γ, the term a reduces to a result v, yielding an updated store t.

As previously argued, the explicit Abadi-Cardelli's stack S disappears from the reduction, thus simplifying the judgment itself and the formal proofs about it. Its content is distributed in Γ, i.e. Γ contains enough assumptions to carry the association between variables and results. These bindings are in the form of hypothetical premises local to sub-reductions, then discharged in the spirit of natural deduction style.

It is worth noticing that we do not introduce the well-formedness judgments for stores and environments: these properties are automatically ensured by the freshness conditions of *eigenvariables* in the natural deduction style.

Closures. An immediate consequence of our approach is that we cannot retain the **imp**ς's closures, i.e. pairs $\langle method, stack \rangle$, because there are no explicit stacks to put in anymore. We yield a different, more efficient handling of closures, by gathering from the environment the results associated to the free variables of methods bodies. The translation we design is the following ($S = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$):

$$\langle \varsigma(x)b, S \rangle \quad \longmapsto \quad \lambda x.(bind\ v_1\ \lambda x_1.(bind \ldots (bind\ v_n\ \lambda x_n.(ground\ b)) \ldots))$$

where the first (outermost) abstraction λx stands for $\varsigma(x)$, i.e. `self`, and the n remaining abstractions ($n \geq 0$) capture the free variables of b. That is, x_i appear free in $(ground\ b)$ and $(x_i, v_i) \in \Gamma\ \forall i \in 1..n$.

For instance, the method m of the object $[m = \varsigma(x)x]$ is stored $\lambda x.(ground\ x)$ and the method n of $let\ y = a\ in\ [n = \varsigma(x)y]$ is stored $\lambda x.(bind\ v_a\ \lambda y.(ground\ y))$, where the term a reduces to v_a. We imagine to carry out this process in two steps: the free variables of a method body are captured by λ-abstractions, then they are bound to suitable results of Γ. The syntax of closures reflects the fact that methods can be eventually ground terms:

$$\begin{array}{rcll} c, d & ::= & \lambda x.ground(b) & \text{ground term} \\ & & \lambda x.bind(v, d) & \text{free variable case} \end{array}$$

where b is a term and v a result of $\mathbf{imp}\varsigma$. So, the entities we use are the following:

$$\begin{array}{rcll} \iota & & & \text{store location} \\ s & ::= & \iota_i \mapsto \lambda x_i.c_i\,^{i\in 1..n} & \text{store} \\ v & ::= & [l_i = \iota_i\,^{i\in 1..n}] & \text{result} \end{array}$$

Our design of closures requires the introduction of an extra reduction predicate, which is needed for evaluating closures —as required by the method selection, that fetches a closure in the store and evaluates it. This reduction is defined by mutual induction with $eval$:

$$eval_{body} \subseteq Store \times Body \times Store \times Res$$

where the sort *Body* stands for *closure bodies*, i.e. terms where the only possibly free variable is the one corresponding to `self`.

Next we have to describe the formation of closures. We adopt the technique used in [BH90, Mic94] for ensuring that a lambda expression carries with it enough information about the results associated with its free variables. First we introduce the predicate:

$$closed \subseteq Term$$

The intended meaning of $(closed\ a)$ is that a does not contain free variables, except the ones assumed to be closed in the context, namely:

$$\Gamma \vdash closed(a) \ \Leftrightarrow\ \text{for all } x \in FV(a).\ closed(x) \in \Gamma$$

We define the predicate *closed* by induction on the syntax of terms; it conveys the binding nature of the constructs of the $\mathbf{imp}\varsigma$-calculus:

$$\text{(c\_obj)}\ \frac{\begin{array}{c}(closed(x_i)) \\ \vdots \\ (closed\ b_i) \quad \forall i \in 1..n\end{array}}{(closed\ [l_i = \varsigma(x_i)b_i\,^{i\in 1..n}])}$$

$$\text{(c\_call)}\ \frac{(closed\ a)}{(closed\ a.l)}$$

$$\text{(c\_over)}\ \frac{\begin{array}{c}(closed(x))\\ \vdots\\ (closed\ a) \quad (closed\ b)\end{array}}{(closed\ (a.l \leftarrow \varsigma(x)b))}$$

$$\text{(c\_clone)}\ \frac{(closed\ a)}{(closed\ (clone\ a))}$$

$$\text{(c\_let)}\ \frac{\begin{array}{c}(closed(x))\\ \vdots\\ (closed\ a) \quad (closed\ b)\end{array}}{(closed\ (let\ x = a\ in\ b))}$$

So doing, we can introduce the core construction of closure bodies:

$$wrap \subseteq Term \times Body$$

The intended meaning of $\Gamma \vdash (wrap\ b\ c)$ is that c is a closure body obtained by binding all free variables in the term b to their respective results in Γ. Intuitively, the rules for *wrap* allow for successively binding the free variables appearing in the method body, until it is "closed". Notice that we write below $(x\ fresh)$ for a variable name not used in the context Γ, and $a\{z/y\}$ for the replacement of the free occurrences of y with z in a:

$$\text{(w\_ground)}\ \frac{(closed\ b)}{(wrap\ b\ (ground\ b))}$$

$$\text{(w\_bind)}\ \frac{\begin{array}{c}(closed(z))\\ \vdots\\ (wrap\ b\{z/y\}\ c\{z/y\}) \quad y \mapsto v \in \Gamma\end{array}}{(wrap\ b\ (bind\ v\ \lambda y.c))}\ (z\ fresh)$$

When we apply rule (*w_bind*), we choose any (free) variable y in b, and we bind it to the corresponding value v, as stated in Γ. The remaining part of the closure c can be seen as the closure body of a method where the variable y is supposed to be "closed", and therefore it is obtained in a proof environment containing this information. This is grasped by the sub-derivation $\Gamma, closed(z) \vdash (wrap\ b\{z/y\}\ c\{z/y\})$, which repeats the rule (*w_bind*) until enough variables have been bound and, correspondingly, enough assumptions of the form $closed(z)$ have been taken to be able to prove $closed(b)$ (i.e. there are no more free variables to bind) and thus apply rule (*w_ground*).

Notice that the closures we get in this manner are "optimized", because only variables which are really free in the body need to be bound in the closure, although in a non-deterministic order.

The *wrap* notion is used in the main term reduction at the moment of reasoning about method closures loaded in the store.

Reduction. We formalize and discuss the two reduction predicates *eval* and $eval_{body}$. The constructors of *eval* correspond to the rules of **imp**ς; the constructors of $eval_{body}$

have not counterpart in the original formulation of the calculus. We adopt here the same notation used by Abadi-Cardelli for extending and updating the store; namely $\sigma, \iota_i \mapsto m_i{}^{i\in 1..n}$ extends σ and $\sigma.\iota_j \leftarrow m$ updates it.

The *variables* are handled trivially; evaluating a variable x produces the same result associated to x in the context, and the store does not change:

$$\text{(e\_var)}\ \frac{x \mapsto v \in \Gamma}{(eval\ s\ x\ s\ v)}$$

We remark that the well-formedness condition in the premise of the rule $(Red\ x)$ disappears, because we have not explicit stacks in our semantics. Notice that we have later to address the well-formedness of the context Γ in such a way that it is coherent with Abadi-Cardelli's notion.

For the sake of presenting the NDS approach without distractions, we consider soon the rule for the *let* construct. Starting from a store s, the rule first evaluates a term a, thus producing a result v and a new store s'. Then it evaluates the term b in the store s' and under the hypothetical premise that a fresh variable x is bound to v; such a premise is discharged in the conclusion of the rule. This is captured by the following natural deduction style rule:

$$\text{(e\_let)}\ \frac{\begin{array}{c}(x \mapsto v)\\ \vdots\\ (eval\ s\ a\ s'\ v) \quad (eval\ s'\ b\ t\ w)\end{array}}{(eval\ s\ (let\ x = a\ in\ b)\ t\ w)}$$

We stress that this rule has to be completely specified, when expressed in a logical framework; in particular, it is necessary to take care that b is an higher-order term and $x \mapsto v$ is a local assumption.

The semantics of *objects* requires the preliminary transformation of the method list, forming the object, into a closure list: this is obtained through the *wrap* notion and the corresponding hypothetical premises. Notice that again the well-formedness condition about the stack disappears, since it is automatically ensured by the natural deduction style.

$$\text{(e\_obj)}\ \frac{\begin{array}{c}(closed(x_i))\\ \vdots\\ (l_i, \iota_i\ distinct) \quad \iota_i \notin dom(\sigma) \quad (wrap\ b_i\ c_i) \quad \forall i \in 1..n\end{array}}{(eval\ s\ [l_i = \varsigma(x_i)b_i{}^{i\in 1..n}]\ (s, \iota_i \mapsto \lambda x.c_i{}^{i\in 1..n})\ [l_i = \iota_i{}^{i\in 1..n}])}$$

Also the *method update* operator needs the *wrap* predicate, because the overrider method has to be preliminary transformed in a closure:

$$\text{(e\_over)}\ \frac{\begin{array}{c}(closed(x))\\ \vdots\\ (eval\ s\ a\ s'\ [l_i = \iota_i{}^{i\in 1..n}]) \quad j \in 1..n \quad \iota_j \in dom(s') \quad (wrap\ b\ c)\end{array}}{(eval\ s\ (a.l \leftarrow \varsigma(x)b)\ (s'.\iota_j \leftarrow \lambda x.c)\ [l_i = \iota_i{}^{i\in 1..n}])}$$

The *clone* operation does not require any specific explanation:

$$\text{(e\_clone)}\ \frac{(eval\ s\ a\ s'\ [l_i = \iota_i{}^{i\in 1..n}])\ \iota_i \in dom(s')\ (\iota'_i\ distinct)\ \iota'_i \notin dom(s')\ \forall i \in 1..n}{(eval\ s\ (clone\ a)\ (s', \iota'_i \mapsto s'(\iota_i)^{i\in 1..n})\ [l_i = \iota'_i{}^{i\in 1..n}])}$$

As previously remarked, the *method selection* requires the evaluation of closure bodies loaded in the store. This evaluation takes place in a context extended with the binding between a fresh variable (representing `self`) and the (implementation of the) host object. Clearly, all the local bindings of the closure have to be unraveled (i.e. assumed in the hypotheses) before the real evaluation of the body is performed. This unraveling is implemented by the auxiliary judgment $eval_{body}$, which can be seen as the dual of *wrap*:

$$\text{(e\_call)}\ \frac{\begin{array}{c}(x \mapsto [l_i = \iota_i{}^{i\in 1..n}])\\ \vdots \\ (eval\ s\ a\ s'\ [l_i = \iota_i{}^{i\in 1..n}]) \quad j \in 1..n \quad s'(\iota_j) = \lambda x.c \quad (eval_{body}\ s'\ c\ t\ w)\end{array}}{(eval\ s\ (a.l_j)\ t\ w)}$$

The predicate $eval_{body}$ is defined by induction on the structure of closures. The base rule corresponds to the case of method bodies whose all free variables are bound to results in Γ, i.e. it is possible to use directly the *eval* predicate. The induction rule evaluates the body of a closure in a context extended with the association between a fresh variable and the value found in the first position of the closure itself:

$$\text{(e\_ground)}\ \frac{(eval\ s\ a\ t\ v)}{(eval_{body}\ s\ (ground\ a)\ t\ v)} \qquad \text{(e\_bind)}\ \frac{\begin{array}{c}(y \mapsto v)\\ \vdots \\ (eval_{body}\ s\ c\ t\ w)\end{array}}{(eval_{body}\ s\ (bind\ v\ \lambda y.c)\ t\ w)}$$

Thus the two evaluation predicates *eval* and $eval_{body}$ are defined by mutual induction. This is the more substantial difference with respect to the original **imp**ς introduced so far. Figures 6.2 and 6.1 collect the rules forming the whole reduction semantics.

Static semantics.

It is immediate to use the natural deduction semantics style for treating also the static semantics. The rephrasing of the type system is less deep and no aspect is new with respect to the case of the operational semantics. The *term typing* judgment is translated as follows:

$$E \vdash a : A \qquad \longmapsto \qquad \Gamma \vdash (type\ a\ A)$$

Thus we formalize the typing of terms by a predicate defined on pairs:

$$type \subseteq Term \times TType$$

where $TType$ stands for the type of terms, i.e. our analogous of Abadi-Cardelli's (object) types. The intended meaning of the derivation of $\Gamma \vdash (type\ a\ A)$ is that, with the knowledge in Γ, the term a types A.

As the stack S disappears from the reduction, the type environment E disappears from the typing, thus simplifying the judgment itself and the formal proofs about it. The global context Γ contains, among other the stuff discussed so far, associations between the free variables x_i, eventually appearing in the object a, and the corresponding types A_i. These typing assignments are used as hypothetical premises about the types of variables, then discharged in the conclusions of rules according to the natural deduction style.

Moreover, as in the case of stacks in the operational semantics, we do not need to introduce a well-formedness judgment for the type environment: typing contexts are automatically ensured to be well-formed by the stack-like discipline of natural deduction and freshness of locally-quantified variables.

Other judgments. The following judgments correspond to the ones used in the original Abadi-Cardelli's setting:

$$\begin{array}{lcl} wt & \subseteq & TType \\ sub & \subseteq & TType \times TType \end{array}$$

The *well-formedness of types* just requires to have distinct labels associated to object types, that must be, in turn, well-formed:

$$(\text{wt\_obj})\ \frac{(wt\ B_i) \quad (l_i\ distinct) \quad \forall i \in 1..n}{(wt\ [l_i : B_i{}^{i\in 1..n}])}$$

The *subtyping* is the smallest reflexive and transitive binary relation on types such that it respects the (*sub_obj*) rule:

$$(\text{sub\_refl})\ \frac{(wt\ A)}{(sub\ A\ A)} \qquad (\text{sub\_trans})\ \frac{(sub\ A\ B) \quad (sub\ B\ C)}{(sub\ A\ C)}$$

$$(\text{sub\_obj})\ \frac{(wt\ B_i) \quad (l_i\ distinct) \quad \forall i \in 1..n+m}{(sub\ [l_i : B_i{}^{i\in 1..n+m}]\ [l_i : B_i{}^{i\in 1..n}])}$$

This rule, telling that longer types are subtypes of shorter ones, captures the fact that the order of the components does not matter and that *invariant*, i.e. neither covariant nor contravariant, types have to correspond to identical labels.

Term typing. We introduce and discuss the constructors of the predicate *type*, comparing them to the original rules of **imp**ς. The *subsumption* is identical to Abadi-Cardelli's one:

$$(\text{t\_sub})\ \frac{(type\ a\ A) \quad (sub\ A\ B)}{(type\ a\ B)}$$

The *variables* are handled as in the case of operational semantics, namely the type of a variable x is the same type associated to x in the context Γ:

$$(\text{t\_var})\ \frac{(wt\ A) \quad x \mapsto A \in \Gamma}{(type\ x\ A)}$$

Notice that, since we do not use the condition of well-formedness of the type environment, we have to assume the well-formedness of the type associated to x. This "hygienic" condition ensures that only well-formed types are assumed in the proof contexts.

The typing of *objects* requires a well-formedness condition in the premise as well:

$$(\text{t\_obj})\ \frac{\begin{array}{c}(x_i \mapsto [l_i : B_i{}^{i\in 1..n}]) \\ \vdots \\ (wt\ [l_i : B_i{}^{i\in 1..n}]) \qquad (type\ b_i\ B_i) \qquad \forall i \in 1..n\end{array}}{(type\ [l_i = \varsigma(x_i)b_i{}^{i\in 1..n}]\ [l_i : B_i{}^{i\in 1..n}])}$$

The remaining typing rules —*method selection*, *method update*, *cloning* and *let*— are simply a faithful translation of the original ones in natural deduction style:

$$\text{(t\_call)}\ \frac{(type\ a\ [l_i : B_i\,^{i\in 1..n}])\quad j \in 1..n}{(type\ (a.l_j)\ B_j)}$$

$$\text{(t\_over)}\ \frac{\begin{array}{c}(x \mapsto [l_i : B_i\,^{i\in 1..n}])\\ \vdots\\ (type\ a\ [l_i : B_i\,^{i\in 1..n}])\quad j \in 1..n \qquad (type\ b\ B_j)\end{array}}{(type\ (a.l \leftarrow \varsigma(x)b)\ [l_i : B_i\,^{i\in 1..n}])}$$

$$\text{(t\_clone)}\ \frac{(type\ a\ [l_i : B_i\,^{i\in 1..n}])}{(type\ (clone\ a)\ [l_i : B_i\,^{i\in 1..n}])}$$

$$\text{(t\_let)}\ \frac{\begin{array}{c}(x \mapsto A)\\ \vdots\\ (type\ a\ A)\quad (type\ b\ B)\end{array}}{(type\ (let\ x = a\ in\ b)\ B)}$$

Figure 6.3 collects the rules forming the whole term typing system.

Example 6.1 *We would like to gain confidence with the new setting of* ***impς****. Let us work with the term let* $y = [l = \varsigma(x)x]$ *in* $[m = \varsigma(x)y]$*: first we evaluate it, thus illustrating the management of closures; after we type it. Let be:*

$$\begin{array}{lcl} s & \equiv & \{0 \mapsto \lambda x.\ (ground\ x)\} \\ t & \equiv & \{0 \mapsto \lambda x.\ (ground\ x),\ 1 \mapsto \lambda x.\ (bind\ [l = 0]\ \lambda y.\ (ground\ y))\} \\ a & \equiv & [l = \varsigma(x)x] \\ b & \equiv & [m = \varsigma(x)y] \\ \mathcal{S} & \equiv & (sub\ [l : [\ ]]\ [\ ]) \end{array}$$

Reduction.

$$\dfrac{\dfrac{\dfrac{(closed(x))_{(1)}}{(wrap\ x\ (grd\ x))}\ (w\_grd)}{(eval\ \emptyset\ a\ s\ [l = 0])}\ (e\_obj)(1) \qquad \dfrac{\dfrac{\dfrac{(closed(z))_{(3)}}{(wrap\ z\ (grd\ z))}\ (w\_grd) \qquad (y \mapsto [l = 0])_{(2)}}{(wrap\ y\ (bind\ [l = 0]\ \lambda y.\ (grd\ y)))}\ (w\_bind)(3)}{(eval\ s\ b\ t\ [m = 1])}\ (e\_obj)}{(eval\ \emptyset\ (let\ y = a\ in\ b)\ t\ [m = 1])}\ (e\_let)(2)$$

Typing.

$$\dfrac{\dfrac{\dfrac{\dfrac{(x \mapsto [l : [\,]])_{(1)}}{(type\ x\ [l : [\,]])}\ (t\_var) \quad \mathcal{S}}{(type\ x\ [\,])}\ (t\_sub)}{(type\ a\ [l : [\,]])}\ (t\_obj)(1) \qquad \dfrac{\dfrac{\dfrac{(y \mapsto [l : [\,]])_{(2)}}{(type\ y\ [l : [\,]])}\ (t\_var) \quad \mathcal{S}}{(type\ y\ [\,])}\ (t\_sub)}{(type\ b\ [m : [\,]])}\ (t\_obj)}{(type\ (let\ y = a\ in\ b)\ [m : [\,]])}\ (t\_let)(2)$$

□

6.3 Adequacy

In this section we compare our formulation of **imp**ς with the original statement of the calculus of chapter 5. We establish in particular the adequacy of the presentation in NDS, which has been carried out for simplifying the formalization in Coq, performed in the next chapter.

Some conventions that we use for addressing the adequacy are the following:

- Var is the syntactical class of variables;

- *dom* is the domain of various structures, as proof contexts, stacks, type environments, stores and store types;

- we write concisely $\Gamma, x \mapsto X \vdash \mathcal{J}$ for natural deduction derivations with hypothetical premise $x \mapsto X$.

First we fix formally the relationship between our heterogeneous context Γ and the environments S, E of Abadi-Cardelli's formulation, and also between the two kinds of stores s and σ in the two approaches.

Definition 6.1 ***(Well-formed context)***
A context Γ is well-formed if:

- *it can be partitioned as $\Gamma = \Gamma_{Res} \cup \Gamma_{TType} \cup \Gamma_{closed}$;*

- *Γ_{Res} contains only formulae of the form $x \mapsto v$, and if $x \mapsto v \in \Gamma_{Res}$, then $x \notin dom(\Gamma_{Res} - \{x\})$;*

- *Γ_{TType} contains only formulae of the form $x \mapsto A$, and if $x \mapsto A \in \Gamma_{TType}$, then $x \notin dom(\Gamma_{TType} - \{x\})$;*

- *Γ_{closed} contains only formulae of the form $closed(x)$, and if $closed(x) \in \Gamma_{closed}$, then $x \notin dom(\Gamma_{closed} - \{x\})$;*

where $x \in Var$, $v \in Res$, $A \in TType$.

Definition 6.2 ***(Environments and stores)***

Let Γ be a context, S a stack, E a type environment, s and σ stores. We define:

$$
\begin{array}{lcl}
\Gamma \subseteq S & \triangleq & \forall x \mapsto v \in \Gamma.\ x \mapsto v \in S \\
\Gamma \subseteq E & \triangleq & \forall x \mapsto A \in \Gamma.\ x{:}A \in E \\
S \subseteq \Gamma & \triangleq & \forall x \mapsto v \in S.\ x \mapsto v \in \Gamma \\
E \subseteq \Gamma & \triangleq & \forall x{:}A \in E.\ x \mapsto A \in \Gamma \\
\gamma(S) & \triangleq & \{x \mapsto S(x) \mid x \mapsto v \in S\} \\
s \lesssim \sigma & \triangleq & \forall \iota_i \in dom(s).\ \gamma(S_i), closed(x_i) \vdash (wrap\ b_i\ s(\iota_i)(x_i)), \\
 & & where\ \sigma(\iota_i) = \langle \varsigma(x_i) b_i, S_i \rangle \\
\sigma \lesssim s & \triangleq & \forall \iota_i \in dom(\sigma).\ \gamma(S_i), closed(x_i) \vdash (wrap\ b_i\ s(\iota_i)(x_i)), \\
 & & where\ \sigma(\iota_i) = \langle \varsigma(x_i) b_i, S_i \rangle
\end{array}
$$

We introduce also two functions which collect, respectively, the bindings and the inner body of a closure; these are defined by recursion:

$$
\begin{array}{ll}
stck(ground(b)) = \emptyset & stck(bind(v, \lambda x.c)) = stck(c) \cup \{x \mapsto v\} \\
body(ground(b)) = b & body(bind(v, \lambda x.c)) = body(c)
\end{array}
$$

Static semantics. We start addressing the adequacy of the static semantics. First we state some properties of the original **imp**ς that are useful for proving the following results.

Lemma 6.1 ***(Properties of typing)***
(i). If $E \vdash a : A$, then $E \vdash \diamond$.
(ii). If $E \vdash A <: B$, then $E \vdash A$ and $E \vdash B$.
(iii). If $E \vdash a : A$, then $E \vdash A$.

Proof. (i). By structural induction on the derivation of $E \vdash a : A$.
(ii). By structural induction on the derivation of $E \vdash A <: B$.
(iii). By structural induction on the derivation of $E \vdash a : A$, points (i) and (ii). □

Now we prove the adequacy of the auxiliary judgments. Using these properties we can establish the soundness and completeness of the main term typing judgment.

Lemma 6.2 ***(Auxiliary typing judgments)***
(i). If $\Gamma \vdash (sub\ A\ B)$, then $\forall E.\ E \vdash A <: B$.
(ii). If $E \vdash A <: B$, then $\forall \Gamma.\ \Gamma \vdash (sub\ A\ B)$.
(iii). If $E \vdash A$, then $\forall \Gamma.\ \Gamma \vdash (wt\ A)$.

Proof. (i). By structural induction on the derivation of $\Gamma \vdash (sub\ A\ B)$.
(ii). By structural induction on the derivation of $E \vdash A <: B$.
(iii). By structural induction on the derivation of $E \vdash A$. □

Theorem 6.1 ***(Soundness of term typing)***
Let be Γ well-formed and E such that $E \vdash \diamond$. If $\Gamma \subseteq E$ and $\Gamma \vdash (type\ a\ A)$, then $E \vdash a : A$.

Proof. By structural induction on the derivation of $\Gamma \vdash (type\ a\ A)$.
(t_sub). By inductive hypothesis, lemma 6.2.(i) and rule (Val Subsumption).
(t_var). By rule (Val x).
(t_obj). By inductive hypothesis and rule (Val Object).

(t_call). By inductive hypothesis and rule (Val Select).
(t_over). By inductive hypothesis and rule (Val Update).
(t_clone). By inductive hypothesis and rule (Val Clone).
(t_let). By inductive hypothesis and rule (Val Let). □

Theorem 6.2 ***(Completeness of term typing)***
Let be Γ *well-formed and* E *such that* $E \vdash \diamond$. *If* $E \subseteq \Gamma$ *and* $E \vdash a : A$, *then* $\Gamma \vdash (type\ a\ A)$.

Proof. By structural induction on the derivation of $E \vdash a : A$.
(Val Subsumption). By inductive hypothesis, lemma 6.2.(ii) and rule (t_sub).
(Val x). By lemma 6.2.(iii) and rule (t_var).
(Val Object). By inductive hypothesis, lemma 6.1.(iii) and rule (t_obj).
(Val Select). By inductive hypothesis and rule (t_call).
(Val Update). By inductive hypothesis and rule (t_over).
(Val Clone). By inductive hypothesis and rule (t_clone).
(Val Let). By inductive hypothesis and rule (t_let). □

Dynamic semantics. The adequacy of the term reduction is addressed similarly to the typing, but the proof is more technical, because it is required to take account of stores.

Lemma 6.3 ***(Adequacy of reduction: auxiliary properties)***
Let be Γ *well-formed and* S *such that* $\sigma \cdot S \vdash \diamond$:

(i). If $s \lesssim \sigma$, $\Gamma \subseteq S$ *and* $\Gamma, closed(x_i) \vdash (wrap\ b_i\ c_i)\ \forall i \in I$, *then* $(s, \iota_i \mapsto \lambda x_i.c_i{}^{i\in I}) \lesssim (\sigma, \iota_i \mapsto \langle \varsigma(x_i)b_i, S\rangle^{i\in I})$.

(ii). If $\sigma \lesssim s$ *and* $\gamma(S), cld(x_i) \vdash (wrap\ b_i\ c_i)\ \forall i \in I$, *then* $(\sigma, \iota_i \mapsto \langle \varsigma(x_i)b_i, S\rangle^{i\in I}) \lesssim (s, \iota_i \mapsto \lambda x_i.c_i{}^{i\in I})$.

(iii). If $s \lesssim \sigma$, $\Gamma \subseteq S$ *and* $\Gamma, closed(x) \vdash (wrap\ b\ c)$, *then* $(s.\iota_i \leftarrow \lambda x.c) \lesssim (\sigma.\iota_i \leftarrow \langle \varsigma(x)b, S\rangle)$.

(iv). If $\sigma \lesssim s$ *and* $\gamma(S), closed(x) \vdash (wrap\ b\ c)$, *then* $(\sigma.\iota_i \leftarrow \langle \varsigma(x)b, S\rangle) \lesssim (s.\iota_i \leftarrow \lambda x.c)$.

(v). If $s \lesssim \sigma$ *and* $\iota_i \in dom(s)\ \forall i \in I$, *then* $(s, \iota'_i \mapsto s(\iota_i)^{i\in I}) \lesssim (\sigma, \iota'_i \mapsto \sigma(\iota_i)^{i\in I})$.

(vi). If $\sigma \lesssim s$ *and* $\iota_i \in dom(\sigma)\ \forall i \in I$, *then* $(\sigma, \iota'_i \mapsto \sigma(\iota_i)^{i\in I}) \lesssim (s, \iota'_i \mapsto s(\iota_i)^{i\in I})$.

(vii). If $\Gamma, closed(x) \vdash (wrap\ b\ c)$ *and* $\Gamma, x \mapsto w \vdash (eval\ s\ b\ t\ v)$, *then* $\Gamma, x \mapsto w \vdash (eval_{body}\ s\ c\ t\ v)$.

Proof. (i) ... (vi). The proofs are immediate by definitions of $s \lesssim \sigma$ and $\sigma \lesssim s$ in 6.2.
(vii). By induction on c. □

Theorem 6.3 ***(Soundness of reduction)***
Let be Γ *well-formed and* S *such that* $\sigma \cdot S \vdash \diamond$. *Let be* $\Gamma \subseteq S$ *and* $s \lesssim \sigma$.
(a). If $\Gamma \vdash (eval\ s\ a\ t\ v)$, *then there exists* τ *such that* $\sigma \cdot S \vdash a \rightsquigarrow v \cdot \tau$ *and* $t \lesssim \tau$.
(b). If $\Gamma \vdash (eval_{body}\ s\ c\ t\ v)$, *then there exists* τ *such that* $\sigma \cdot stck(c) \vdash body(c) \rightsquigarrow v \cdot \tau$ *and* $t \lesssim \tau$.

Proof. By mutual structural induction on the derivation of $\Gamma \vdash (eval\ s\ a\ t\ v)$ and $\Gamma \vdash (eval_{body}\ s\ c\ t\ v)$.
(e_var). By rule (Red x).
(e_obj). By rule (Red Object) and lemma 6.3.(i).

(e_call). By inductive hypothesis, mutual inductive hypothesis and rule (Red Select).
(e_over). By inductive hypothesis, rule (Red Update) and lemma 6.3.(iii).
(e_clone). By inductive hypothesis, rule (Red Clone) and lemma 6.3.(v).
(e_let). By inductive hypothesis and rule (Red Let).
(e_ground). By mutual induction.
(e_bind). By inductive hypothesis. □

Theorem 6.4 ***(Completeness of reduction)***
Let be Γ *well-formed and* S *such that* $\sigma \cdot S \vdash \diamond$. *Let be* $S \subseteq \Gamma$ *and* $\sigma \lesssim s$.
If $\sigma \cdot S \vdash a \rightsquigarrow v \cdot \tau$, *then there exists* t *such that* $\Gamma \vdash (eval\ s\ a\ t\ v)$ *and* $\tau \lesssim t$.

Proof. By structural induction on the derivation of $\sigma \cdot S \vdash a \rightsquigarrow v \cdot \tau$.
(Red x). By rule (e_var).
(Red Object). First derive $\gamma(S), closed(x_i) \vdash (wrap\ b_i\ c_i)\ \forall i \in I$ using the rules for *wrap*. Then take $t := s, \iota_i \mapsto \lambda x_i.c_i\,^{i\in I}$ and conclude by rule (e_obj) and lemma 6.3.(ii).
(Red Select). By inductive hypothesis (twice), lemma 6.3.(vii) and rule (e_call).
(Red Update). Apply the inductive hypothesis (thus obtaining s'), then derive $\gamma(S), closed(x) \vdash (wrap\ b\ c)$ using the rules for *wrap*. Take $t := s', \iota_j \leftarrow \lambda x.c$ and conclude by rule (e_over) and lemma 6.3.(iv).
(Red Clone). By inductive hypothesis, rule (e_clone) and lemma 6.3.(vi).
(Red Let). By inductive hypothesis and rule (e_let). □

6.4 Preliminary metatheory

We establish in this section some properties of the static semantics which are useful for proving the Subject Reduction theorem. These results relate the term typing judgment to the subtyping and the well-formedness of types.

Lemma 6.4 ***(Properties of the typing system)***
Let Γ *be a well-formed context. Then:*

$$(var). \quad : \quad \Gamma \vdash (type\ x\ A)\ \Rightarrow\ \exists B : TType.\ (x \mapsto B \in \Gamma)\ \wedge\ \Gamma \vdash (sub\ B\ A)$$

$$(obj). \quad : \quad \Gamma \vdash (type\ [l_i = \varsigma(x_i)b_i\,^{i\in 1..n}]\ A)\ \Rightarrow\ \exists B : TType.\ \Gamma \vdash (type\ [l_i = \varsigma(x_i)b_i\,^{i\in 1..n}]\ B)\ \wedge\ \Gamma \vdash (B <: A)$$

$$(call). \quad : \quad \Gamma \vdash (type\ a.l\ A)\ \Rightarrow\ \exists B, B_l : TType.\ \Gamma \vdash (type\ a\ B)\ \wedge\ (l, B_l) \in B\ \wedge\ \Gamma \vdash (sub\ B_l\ A)$$

$$(over). \quad : \quad \Gamma \vdash (type\ (a.l \leftarrow \varsigma(x)b)\ A)\ \Rightarrow\ \exists B, B_l : TType.\ \Gamma \vdash (type\ a\ B)\ \wedge\ (l, B_l) \in B\ \wedge\ \Gamma \vdash (sub\ B\ A)\ \wedge\ \Gamma, x \mapsto B \vdash (type\ b\ B_l)$$

$$(clone). \quad : \quad \Gamma \vdash (type\ (clone\ a)\ A)\ \Rightarrow\ \exists B : TType.\ \Gamma \vdash (type\ a\ B)\ \wedge\ \Gamma \vdash (sub\ B\ A)$$

$$(let). \quad : \quad \Gamma \vdash (type\ (let\ x = a\ in\ b)\ A)\ \Rightarrow\ \exists B, C : TType. \\ \Gamma \vdash (type\ a\ C)\ \wedge\ \Gamma \vdash (sub\ B\ A)\ \wedge\ \Gamma, x \mapsto C \vdash (type\ b\ B)$$

$$(sub{-}wt). \quad : \quad \Gamma \vdash (sub\ A\ B)\ \Rightarrow\ \Gamma \vdash (wt\ A)\ \wedge\ \Gamma \vdash (wt\ B)$$

$$(type{-}wt). \quad : \quad \Gamma \vdash (type\ a\ A)\ \Rightarrow\ \Gamma \vdash (wt\ A)$$

$$(bd{-}weak). \quad : \quad \Gamma, x \mapsto A \vdash (type\ b\ C)\ \wedge\ (sub\ B\ A)\ \Rightarrow \\ \Gamma, x \mapsto B \vdash (type\ b\ C)$$

Proof. All the statements are proved by structural induction on the (first) hypothesis. □

6.5 Result typing by coinduction

Since results contain references to the store, we need to type store locations in order to be able to type results. In principle, the type of a store location is the type of its content, which in turn may contain results, i.e. other pointers to the store. Therefore, this potential presence of loops makes the typing system for results non-trivial: a naive system would unravel the pointers indefinitely, chasing non well-founded structures in the memory.

We have already remarked that the solution adopted by Abadi-Cardelli (see section 5.3) is to introduce yet another typing structure, i.e. *store types*, which assign to each store location a type consistent with the content of the location. Store types have to be provided beforehand, and suitable auxiliary judgments are needed in order to assign types to results and checking the consistency of store types with stores. Clearly, all this technical machinery adds further complexity to the (already difficult, though) metatheory of the system.

Thus we have devised an alternative formalization, inspired by the features of $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$, where the canonical way for dealing with non well-founded, circular data is *coinduction*. The idea at the heart of our system is simple: in order to check whether a result can be given a type A, we open all the closures pointed to and follow, in turn, the pointers belonging to the closures, thus visiting the store until a closure without pointers is reached; in such a case we can type a closure using the traditional *type* judgment. If the original pointer is encountered in the meanwhile, the type A we started with is used. Clearly, this means that the predicate we are proving has to be *assumed* in the hypotheses, hence the coinduction.

We are formalizing our approach, which adheres to these considerations. We will see that our attempt is quite successful for the fragment of $\mathbf{imp}\varsigma$ without the "method update" operator, although is not sufficient for the full calculus. So we will develop as well in the next section an encoding which adheres to Abadi-Cardelli's setting.

Coinductive result typing. Abadi-Cardelli's *result typing* and *store typing* judgments collapse into an unique judgment:

$$\Sigma \models v : A, \quad \Sigma \models \sigma \qquad \longmapsto \qquad \Gamma \vdash (cores\ s\ v\ A)$$

Namely, we introduce a predicate *cores* defined on triples:

$$cores\ \subseteq\ Store \times Res \times TType$$

The intended meaning of the derivation of $\Gamma \vdash (cores\ s\ v\ A)$ is that, using the assumptions in Γ, the result v types A w.r.t. the store s.

Formalization. The typing of results requires to type closure bodies pointed to in the store. This forces to introduce a specialized predicate:

$$cotype_{body} \subseteq Store \times Body \times TType$$

This solution is similar to the case of the $eval_{body}$ predicate, used in the operational semantics. The intended meaning of $\Gamma \vdash (cotype_{body}\ s\ b\ A)$ is that, using Γ, the method body b types A with respect to the store s it belongs to. Notice that this judgment arises just in sub-derivations of the main *cores* proof system.

The predicates *cores* and $cotype_{body}$ are formalized by means of a mutual definition, where the former is coinductive and the latter inductive:

$$\text{(t\_cores)}\ \frac{\begin{array}{ccccc} & & (\mathcal{J}),\ (x \mapsto [l_i : B_i\,^{i\in 1..n}]) & & \\ (wt\ [l_i : B_i\,^{i\in 1..n}]) & & \vdots & & (distinct\ \iota_i) \\ s(\iota_i) = \lambda x_i.c_i & & (cotype_{body}\ s\ c_i\ B_i) & & \iota_i \in dom(s) \quad \forall i \in 1..n \end{array}}{(cores\ s\ [l_i = \iota_i\,^{i\in 1..n}]\ [l_i : B_i\,^{i\in 1..n}])}$$

$$\text{(t\_coground)}\ \frac{(type\ b\ A)}{(cotype_{body}\ s\ (ground\ b)\ A)}$$

$$\text{(t\_cobind)}\ \frac{\begin{array}{cc} & (y \mapsto A) \\ & \vdots \\ (cores\ s\ v\ A) & (cotype_{body}\ s\ c\ B) \end{array}}{(cotype_{body}\ s\ (bind\ v\ \lambda y.c)\ B)}$$

where $\mathcal{J} \equiv (cores\ s\ [l_i = \iota_i\,^{i\in 1..n}]\ [l_i : B_i\,^{i\in 1..n}])$. The rule (*t_cores*), provided the pointers are distinct and not dangling, fetches the right tuple of closures and delegates their typing to the predicate $cotype_{body}$, in a context enriched by an hypothetical premise which binds a fresh variable to the whole result type. Notice that the result type has to be well-formed.

The rules for the $cotype_{body}$ predicate are given by cases on the structure of closure bodies. The rule (*t_coground*) simply delegates the typing of a "ground" body to the *type* judgment; (*t_cobind*) requires both to type the first result appearing in the body —through *cores*— and to type recursively the sub-body with the type of the result in the scope.

As the reader can see, the typing system for results is simple and compact, essentially because we do not need store types (and all related machinery) anymore. Also, since stacks and type environments are distributed in the proof contexts, we do not need *stack typing* judgment either; however, in stating and proving the Subject Reduction theorem we will require that the types associated to variables in the context are consistent with the results associated to same variables. Figure 6.4 reports the coinductive result typing system.

Examples. It is important to remark that the rule for the predicate *cores* is *potentially* coinductive. That is, depending on particular stores, it can be used infinitely many times

—handling stores with cycles— or finitely many —in the case of stores without cycles. This difference is reflected by the use of the coinductive hypothesis.

Let us detail an example. First we carry out an inductive proof: we type the result $[m = 1]$ w.r.t. the store without cycles t used in the term typing example 6.1. After we type the result $[l = 0]$ w.r.t. a store with a loop, thus reasoning by coinduction. Let be:

$$\begin{array}{lcl} t & \equiv & \{0 \mapsto \lambda x.\ (ground\ x),\ 1 \mapsto \lambda x.\ (bind\ [l = 0]\ \lambda y.\ (ground\ y))\} \\ u & \equiv & \{0 \mapsto \lambda x.(bind\ [l = 0]\ \lambda y.(ground\ y))\} \\ \mathcal{S} & \equiv & (sub\ [l : [\ ]]\ [\ ]) \end{array}$$

The inductive proof is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(x \mapsto [l : [\ ]])_{(3)}}{(type\ x\ [l : [\ ]])}\ (t\_var) \quad \mathcal{S}}{(type\ x\ [\ ])}\ (t\_sub)}{(cotype_{body}\ t\ (grd\ x))\ [\ ])}\ (t\_cgrd)}{(cores\ t\ [l = 0]\ [l : [\ ]])}\ (t\_cores)(3) \quad \dfrac{\dfrac{\dfrac{\dfrac{(y \mapsto [l : [\ ]])_{(2)}}{(type\ y\ [l : [\ ]])}\ (t\_var) \quad \mathcal{S}}{(type\ y\ [\ ])}\ (t\_sub)}{(cotype_{body}\ t\ (grd\ y))\ [\ ])}\ (t\_cgrd)}{\dfrac{(cotype_{body}\ t\ (bind\ [l = 0]\ \lambda y.(grd\ y))\ [\ ])}{(cores\ t\ [m = 1]\ [m : [\ ]])}\ (t\_cores)(1)}\ (t\_cobind)(2)$$

The coinductive proof is the following:

$$\dfrac{\dfrac{(cores\ u\ [l = 0]\ [l : [\ ]])_{(1)} \quad \dfrac{\dfrac{\dfrac{(y \mapsto [l : [\ ]])_{(2)}}{(type\ y\ [l : [\ ]])}\ (t\_var) \quad \mathcal{S}}{(type\ y\ [\ ])}\ (t\_sub)}{(cotype_{body}\ u\ (ground\ y)\ [\ ])}\ (t\_coground)}{(cotype_{body}\ u\ (bind\ [l = 0]\ \lambda y.(ground\ y))\ [\ ])}\ (t\_cobind)(2)}{(cores\ u\ [l = 0]\ [l : [\ ]])}\ (t\_cores)(1)$$

Adequacy. Since we use coinductive proof systems, our perspective is quite different from the original formulation of **imp**ς. Anyway, we state the following adequacy result.

Theorem 6.5 ***(Soundness of coinductive result typing)***
Let be Γ *well-formed and* $s \lesssim \sigma$.
If $\Gamma \vdash (cores\ s\ v\ A)$, *then there exists* Σ *such that* $\Sigma \models v : A$ *and* $\Sigma \models \sigma$.

Proof. By inspection on the rules (t_cores), $(t\_coground)$, (t_cobind) and theorem 6.1. □

Theorem 6.6 ***(Completeness of coinductive result typing)***
Let be Γ *well-formed and* $\sigma \lesssim s$.
If $\Sigma \models v : A$ *and* $\Sigma \models \sigma$, *then* $\Gamma \vdash (cores\ s\ v\ A)$.

Proof. By inspection on the derivations of $\Sigma \models v : A$, $\Sigma \models \sigma$ and theorem 6.2. □

Subject Reduction. We present the proof of the Subject Reduction for the **impς**-calculus without the "method update" construct: the theorem is simplified both in the statement and the proof. It is currently an open problem to extend the proof to the full calculus.

Some preliminary work address the formal development of the metatheory about the result typing and specific properties of the various operators of the calculus. In the following Γ is a well-formed context that, when clear from the text, is omitted from judgments.

Lemma 6.5 ***(Coinductive result typing)***

$$(i). \quad : \quad (cores\ s\ v\ A)\ \wedge\ (eval\ s\ a\ t\ w)\ \Rightarrow\ (cores\ t\ v\ A)$$

$$(ii). \quad : \quad \Gamma, x \mapsto A \vdash (type\ b\ B)\ \wedge\ \Gamma, closed(x) \vdash (wrap\ b\ c)\ \wedge\ s(\iota) = \lambda x.c\ \wedge\ (\forall x, w, C.\ (x \mapsto w, x \mapsto C \in \Gamma) \Rightarrow \Gamma \vdash (cores\ s\ w\ C))\ \Rightarrow\ \Gamma, x \mapsto A \vdash (cotype_{body}\ s\ c\ B)$$

Proof. (i). By structural induction on the derivation of $\Gamma \vdash (eval\ s\ a\ t\ w)$, because the store s cannot be updated.

(ii). By structural induction on the derivation of $\Gamma, closed(x) \vdash (wrap\ b\ c)$. □

Lemma 6.6 ***(Objects)***

$$(i). \quad : \quad (type\ [l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}]\ [l_i : B_i\ ^{i\in 1..n}])\ \wedge\ \Gamma, cld(x_i) \vdash (wrap\ b_i\ c_i)\ ^{i\in 1..n}\ \wedge\ (\forall x, w, C.\ (x \mapsto w, x \mapsto C \in \Gamma) \Rightarrow \Gamma \vdash (cores\ s\ w\ C))\ \Rightarrow\ (cores\ (s, \iota_i \mapsto c_i\ ^{i\in 1..n})\ [l_i = \iota_i\ ^{i\in 1..n}]\ [l_i : B_i\ ^{i\in 1..n}])$$

Proof. (i). By induction on the object $[l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}]$ and lemma 6.5.(ii). □

Lemma 6.7 ***(Select)***

$$(i). \quad : \quad (cores\ s\ [l_j : \iota_j, \ldots]\ [l_j : B_j, \ldots])\ \wedge\ s(\iota_j) = \lambda x.c\ \Rightarrow\ \Gamma, x \mapsto [l_j : B_j, \ldots] \vdash (cotype_{body}\ s\ c\ B_j)$$

Proof. (i). By induction on the object type $[l_j : B_j, \ldots]$. □

Lemma 6.8 ***(Clone)***

$$(i). \quad : \quad (cores\ s\ v\ A)\ \Rightarrow\ (cores\ (s, t)\ v\ A)$$

$$(ii). \quad : \quad (cores\ s\ [l_i = \iota_i\ ^{i\in 1..n}]\ A)\ \Rightarrow\ (cores\ (s, \iota'_i \mapsto s(\iota_i)\ ^{i\in 1..n})\ [l_i = \iota'_i\ ^{i\in 1..n}]\ A)$$

Proof. (i). By coinduction.

(ii). By induction on the result $[l_i = \iota_i\ ^{i\in 1..n}]$. □

We are now ready for stating and proving the Subject Reduction result. Due to the rephrasing of **imp**ς, we have to ask for the coherence between types and results associated to variables in the context; that is, given the store s:

$$\forall x, w, C.\ x \mapsto w,\ x \mapsto C \in \Gamma \ \Rightarrow\ \Gamma \vdash (cores\ s\ w\ C)$$

This corresponds to the (Store Typing) Abadi-Cardelli's judgment, but our management, through distributed stacks and environments, is easier. By this rearrangement, we obtain the following statement of the Subject Reduction theorem, which is simpler than Abadi-Cardelli's one, thus leading to a less complex proof.

Theorem 6.7 ***(Subject Reduction, coinductive setting, no method update)*** *Let Γ be a well-formed context. Then:*

$$\begin{array}{l}\Gamma \vdash (type\ a\ A)\ \wedge\ \Gamma \vdash (eval\ s\ a\ t\ v)\ \wedge \\ (\forall x, w, C.\ (x \mapsto w, x \mapsto C \in \Gamma) \Rightarrow \Gamma \vdash (cores\ s\ w\ C))\ \Rightarrow \\ \exists A^+ : TType. \\ \Gamma \vdash (cores\ t\ v\ A^+)\ \wedge\ \Gamma \vdash (sub\ A^+\ A)\end{array}$$

Proof. By structural induction on the derivation of $\Gamma \vdash (eval\ s\ a\ t\ v)$. The rules *e_call* and *e_bind* require a *mutual* structural induction argument, namely a stronger induction schema valid also for the predicate $eval_{body}$, which is the counterpart of *eval* for closures. We present the proof by omitting the context Γ, that is, $\mathcal{J}$ stands for $\Gamma \vdash \mathcal{J}$.

(e_var). By hypothesis $(type\ x\ A)$ and:

$$(\text{e\_var})\ \frac{x \mapsto v \in \Gamma}{(eval\ s\ x\ s\ v)}$$

From lemma 6.4.(var), there exists B such that $x \mapsto B \in \Gamma$ and $(sub\ B\ A)$. Choose $A^+ := B$.

Since $x \mapsto v \in \Gamma$, by the 3rd hypothesis of the theorem we can derive $(cores\ s\ v\ A^+)$, thus concluding.

(e_obj). By hypothesis $(type\ [l_i = \varsigma(x_i)b_i{}^{i\in 1..n}]\ A)$ and:

$$(\text{e\_obj})\ \frac{\begin{array}{c}(closed(x_i)) \\ \vdots \\ (l_i, \iota_i\ distinct) \quad \iota_i \notin dom(\sigma) \quad (wrap\ b_i\ c_i) \quad \forall i \in 1..n\end{array}}{(eval\ s\ [l_i = \varsigma(x_i)b_i{}^{i\in 1..n}]\ (s, \iota_i \mapsto \lambda x.c_i{}^{i\in 1..n})\ [l_i = \iota_i{}^{i\in 1..n}])}$$

By lemma 6.4.(obj), there exists $[l_i : B_i{}^{i\in 1..n}]$ such that:

$$(type\ [l_i = \varsigma(x_i)b_i{}^{i\in 1..n}]\ [l_i : B_i{}^{i\in 1..n}]) \tag{6.1}$$

$$(sub\ [l_i : B_i{}^{i\in 1..n}]\ A) \tag{6.2}$$

Choose $A^+ := [l_i : B_i{}^{i\in 1..n}]$.

Since $\Gamma, closed(x_i) \vdash (wrap\ b_i\ c_i)\ \forall i \in 1..n$ and 6.1, we apply the lemma 6.6.(i), thus deducing $(cores\ (s, \iota_i \mapsto \lambda x_i.c_i{}^{i\in 1..n})\ [l_i = \iota_i{}^{i\in 1..n}]\ A^+)$; we conclude by 6.2.

(e_call). By hypothesis $(type\ (a.l_j)\ A)$ and:

$$\text{(e\_call)}\ \frac{\begin{array}{c}(x \mapsto [l_i = \iota_i\,^{i\in 1..n}])\\ \vdots\\ (eval\ s\ a\ s'\ [l_i = \iota_i\,^{i\in 1..n}]) \quad j \in 1..n \quad s'(\iota_j) = \lambda x.c \quad (eval_{body}\ s'\ c\ t\ w)\end{array}}{(eval\ s\ (a.l_j)\ t\ w)}$$

By lemma 6.4.(call), there exists $[l_j : B_j, \ldots]$ such that $(type\ a\ [l_j : B_j, \ldots])$ and $(sub\ B_j\ A)$. Since $(eval\ s\ a\ s'\ [l_i = \iota_i\,^{i\in 1..n}])$, by inductive hypothesis there exists C such that:
(a). $(cores\ s'\ [l_i = \iota_i\,^{i\in 1..n}]\ C)$;
(b). $(sub\ C\ [l_j : B_j, \ldots])$.
By the rule (e_call), it is $j \in 1..n$, $s'(\iota_j) = \lambda x.c$ and:

$$\Gamma, x \mapsto [l_i = \iota_i\,^{i\in 1..n}] \vdash (eval_{body}\ s'\ c\ t\ w) \tag{6.3}$$

On the other hand, we have $C \equiv [l_j : B_j, \ldots]$ from (b), thus, using (a) and lemma 6.7.(i):

$$\Gamma, x \mapsto C \vdash (type_{body}\ s'\ c\ B_j) \tag{6.4}$$

We can deduce $(\forall x, w, C.\ (x \mapsto w, x \mapsto C \in \Gamma) \Rightarrow \Gamma \vdash (cores\ s'\ w\ C))$ from the third hypothesis of the theorem and lemma 6.5.(i); therefore, since 6.3 and 6.4, we can apply the mutual induction hypothesis, deducing there exists A^+ such that $(cores\ t\ w\ A^+)$ and $(sub\ A^+\ B_j)$. We finish by transitivity of subtyping.

(e_clone). By hypothesis $(type\ (clone\ a)\ A)$ and:

$$\text{(e\_clone)}\ \frac{(eval\ s\ a\ s'\ [l_i = \iota_i\,^{i\in 1..n}])\ \iota_i \in dom(s')\ (\iota'_i\ distinct)\ \iota'_i \notin dom(s')\ \forall i \in 1..n}{(eval\ s\ (clone\ a)\ (s', \iota'_i \mapsto s'(\iota_i)\,^{i\in 1..n})\ [l_i = \iota'_i\,^{i\in 1..n}])}$$

By lemma 6.4.(clone), there exists B such that $(type\ a\ B)$ and $(sub\ B\ A)$. Since $(eval\ s\ a\ s'\ [l_i = \iota_i\,^{i\in 1..n}])$, we can apply the inductive hypothesis, thus deducing there exists C such that:
(a). $(cores\ s'\ [l_i = \iota_i\,^{i\in 1..n}]\ C)$;
(b). $(sub\ C\ B)$.
Choose $A^+ := C$. We deduce $(sub\ A^+\ A)$ by transitivity of subtyping. We conclude $(cores\ (s', \iota'_i \mapsto s'(\iota_i)^{i\in 1..n})\ [l_i = \iota'_i\,^{i\in 1..n}]\ A^+)$ from (a) and lemma 6.8.(ii).

(e_let). By hypothesis $(type\ (let\ x = a\ in\ b)\ A)$ and:

$$\text{(e\_let)}\ \frac{\begin{array}{c}(x \mapsto v)\\ \vdots\\ (eval\ s\ a\ s'\ v) \quad (eval\ s'\ b\ t\ w)\end{array}}{(eval\ s\ (let\ x = a\ in\ b)\ t\ w)}$$

By lemma 6.4.(let), there exist B, C such that $(type\ a\ C)$ and $\Gamma, x \mapsto C \vdash (type\ b\ B)$ and $(sub\ B\ A)$. Since $(eval\ s\ a\ s'\ v)$, by inductive hypothesis there exists D such that:
(a). $(cores\ s'\ v\ D)$;
(b). $(sub\ D\ C)$.
Since $\Gamma, x \mapsto C \vdash (type\ b\ B)$ and (b), we use lemma 6.4.(bd-weak) for deriving $\Gamma, x \mapsto D \vdash$

$(type\ b\ B)$. Next we deduce $(\forall x, w, C.\ (x \mapsto w, x \mapsto C \in \Gamma) \Rightarrow \Gamma \vdash (cores\ s'\ w\ C))$ from the third hypothesis of the theorem and lemma 6.5.(i). Because of $\Gamma, x \mapsto v \vdash (eval\ s'\ b\ t\ w)$, we apply again the induction hypothesis, thus obtaining E such that $(cores\ t\ v\ E)$ and $(sub\ E\ B)$. Choose $A^+ := E$ and conclude by transitivity of subtyping.

(e_ground). By hypothesis $(cotype_{body}\ s\ (ground\ a)\ A)$ and:

$$\text{(e\_ground)}\ \frac{(eval\ s\ a\ t\ v)}{(eval_{body}\ s\ (ground\ a)\ t\ v)}$$

The assertion $(cotype_{body}\ s\ (ground\ a)\ A)$ has to be derived by the rule $(t\_coground)$, namely from $(type\ a\ A)$: therefore, by induction, there exists A^+ s.t. $(cores\ t\ v\ A^+)$ and $(sub\ A^+\ A)$.

(e_bind). By hypothesis $(cotype_{body}\ s\ (bind\ v\ \lambda y.c)\ A)$ and:

$$\text{(e\_bind)}\ \frac{\begin{array}{c}(y \mapsto v)\\ \vdots \\ (eval_{body}\ s\ c\ t\ w)\end{array}}{(eval_{body}\ s\ (bind\ v\ \lambda y.c)\ t\ w)}$$

The assertion $(cotype_{body}\ s\ (bind\ v\ \lambda y.c)\ A)$ has to be derived by the rule (t_cobind), namely there exists B such that $(cores\ s\ v\ B)$ and $\Gamma, y \mapsto B \vdash (cotype_{body}\ s\ c\ A)$. Therefore, by mutual induction, there exists A^+ such that $(cores\ t\ w\ A^+)$ and $(sub\ A^+\ A)$. □

6.6 Result typing by induction

In this section we scale up to the full **imp**ς, that is, we consider also the "method update" operator. This apparently slight difference yields some deep changes in the typing system for results, because forces to introduce new syntactic structures and the typing system is not coinductive anymore. Consequently, the metatheory about the calculus need to be reformulated; in spite of this, most of the development carried out in the previous section for **imp**ς without update can be readily recovered for the full **imp**ς, thus enlightening the modularity of our development.

Thus we carry out an encoding by following Abadi-Cardelli's approach, that introduce store types in order to type results. The *result typing* judgment changes as follows:

$$\Sigma \models v : A \qquad \longmapsto \qquad \Gamma \vdash (res\ \Sigma\ v\ A)$$

That is, we model a predicate res defined on triples:

$$res \subseteq SType \times Res \times TType$$

where $SType$ stands for the sort of stores, namely collections of method types (see section 5.3). The intended meaning of the derivation $\Gamma \vdash (res\ \Sigma\ v\ A)$ is that, "copying" from the store type Σ, the result v types A. That is, for all $\iota_i \in \pi_2(v)$: $A = \Sigma_1(\iota_i)$.

Next we rephrase the *store typing* and the *extension* relation on store types:

$$\begin{array}{lll}\Sigma \models \sigma & \longmapsto & \Gamma \vdash (comp\ \Sigma\ s)\\ \Sigma' \geq \Sigma & \longmapsto & \Gamma \vdash (ext\ \Sigma'\ \Sigma)\end{array}$$

In the formal development of the metatheory of **imp**$_\varsigma$ we are interested only in stores whose content is "compatible" with store types. If $\Gamma \vdash (comp\ \Sigma\ s)$, then the content of each location in the store s can be given the type indicated by Σ. The extension relation means that store types can only be extended, so it is forbidden to update the contents of their locations.

Formalization. We present the encoding of the predicates *res*, *ext* and *comp*, outlining the differences with respect to the original formulation of **imp**$_\varsigma$.

The rule for typing results copies the type from the store type, with the additional hypotheses that the inferred type is well-formed and that pointers are distinct and not dangling. Since we build store types from (well-formed) types, there is no need to introduce the well-formedness judgment for store types:

$$\text{(t\_res)}\ \frac{(distinct\ \iota_i) \quad \iota_i \in dom(\Sigma) \quad \forall i \in 1..n \qquad \Sigma_1(\iota_i) \equiv [l_i : \Sigma_2(\iota_i)^{\,i\in 1..n}] \quad (wt\ [l_i : \Sigma_2(\iota_i)^{\,i\in 1..n}])}{(res\ \Sigma\ [l_i = \iota_i{}^{\,i\in 1..n}]\ [l_i : \Sigma_2(\iota_i)^{\,i\in 1..n}])}$$

The extension relation between store types simply requires that the greater store type coincides with the smaller one, when restricted to its domain:

$$\text{(t\_ext)}\ \frac{(dom\ \Sigma) \subseteq (dom\ \Sigma') \qquad \forall \iota_i \in (dom\ \Sigma).\ \Sigma'(\iota_i) = \Sigma(\iota_i)}{(ext\ \Sigma'\ \Sigma)}$$

Finally, the compatibility relation between stores and store types requires to introduce a specialized predicate for typing closure bodies:

$$type_{body} \subseteq SType \times Body \times TType$$

This solution is completely similar to the coinductive treatment of previous section; the predicate $type_{body}$ is formalized by induction:

$$\text{(t\_ground)}\ \frac{(type\ b\ A)}{(type_{body}\ \Sigma\ (ground\ b)\ A)} \qquad \text{(t\_bind)}\ \frac{(res\ \Sigma\ v\ A) \quad \begin{array}{c}(y \mapsto A)\\ \vdots\\ (type_{body}\ \Sigma\ c\ B)\end{array}}{(type_{body}\ \Sigma\ (bind\ v\ \lambda y.c)\ B)}$$

So doing, the compatibility can be encoded as follows:

$$\text{(t\_comp)}\ \frac{(dom\ s) \subseteq (dom\ \Sigma) \quad s(\iota_i) = \lambda x_i.c_i \quad \begin{array}{c}(x_i \mapsto \Sigma_1(\iota_i))\\ \vdots\\ (type_{body}\ c_i\ \Sigma_2(\iota_i))\end{array} \quad \forall \iota_i \in dom(s)}{(comp\ \Sigma\ s)}$$

An important difference of this system w.r.t. the corresponding coinductive one of the previous section, is that the rules are in the usual inductive setting. We do not need the coinductive approach, because we do not check the types of locations by chasing pointers in the store; instead, we resort to the store type Σ, which is a finite structure. Clearly,

store types have to be given beforehand; that is, they cannot be synthesized by the typing system.

Finally remember that we do not need the *stack typing* judgment either —in fact, we do not have explicit stacks and type environments at all. However, the correspondence between results and types associated to the same variable in the proof environment will be taken into account in the statement of Subject Reduction theorem. Figure 6.5 reports the inductive result typing system.

Adequacy. We prove the adequacy of the reformulation: notice that it holds also for the calculus without method update, because the full calculus is a conservative extension of it.

Theorem 6.8 ***(Soundness of inductive result typing)***
Let be Γ *well-formed and* Σ *such that* $\Sigma \vdash \diamond$.
For $s \lesssim \sigma$, *if* $\Gamma \vdash (res\ \Sigma\ v\ A)$ *and* $\Gamma \vdash (comp\ \Sigma\ s)$, *then* $\Sigma \models v : A$ *and* $\Sigma \models \sigma$.

Proof. By inspection on the derivations of $\Gamma \vdash (res\ \Sigma\ v\ A)$ and $\Gamma \vdash (comp\ \Sigma\ s)$ and theorem 6.1. □

Theorem 6.9 ***(Completeness of inductive result typing)***
Let be Γ *well-formed and* Σ *such that* $\Sigma \vdash \diamond$.
For $\sigma \lesssim s$, *if* $\Sigma \models v : A$ *and* $\Sigma \models \sigma$, *then* $\Gamma \vdash (res\ \Sigma\ v\ A)$ *and* $\Gamma \vdash (comp\ \Sigma\ s)$.

Proof. By inspection on the derivations of $\Sigma \models v : A$ and $\Sigma \models \sigma$ and theorem 6.2. □

Subject Reduction. As for the coinductive setting of previous section we have to establish a preliminary work about the result typing and some specific properties of the various operators of the calculus. In the following Γ is a well-formed context that is omitted from judgments.

Lemma 6.9 ***(Inductive result typing)***

(i). : $\Gamma \vdash (ext\ \Sigma\ \Sigma)$

(ii). : $\Gamma \vdash (ext\ \Sigma''\ \Sigma') \ \wedge\ \Gamma \vdash (ext\ \Sigma'\ \Sigma) \ \Rightarrow$
$\Gamma \vdash (ext\ \Sigma''\ \Sigma)$

(iii). : $\Gamma \vdash (res\ \Sigma\ v\ A) \ \wedge\ \Gamma \vdash (ext\ \Sigma'\ \Sigma) \ \Rightarrow$
$\Gamma \vdash (res\ \Sigma'\ v\ A)$

(iv). : $\Gamma, x \mapsto A \vdash (type\ b\ B) \ \wedge\ \ \Gamma, closed(x) \vdash (wrap\ b\ c) \ \wedge\ s(\iota) = \lambda x.c \ \wedge$
$(\forall x, w, C.\ (x \mapsto w, x \mapsto C \in \Gamma) \Rightarrow \Gamma \vdash (res\ \Sigma\ w\ C)) \ \Rightarrow$
$\Gamma, x \mapsto A \vdash (type_{body}\ \Sigma\ c\ B)$

Proof. (i), (ii). The proofs are immediate.
(iii). By structural induction on the derivation of $\Gamma \vdash (res\ \Sigma\ v\ A)$.
(iv). By structural induction on the derivation of $\Gamma, closed(x) \vdash (wrap\ b\ c)$. □

Lemma 6.10 ***(Objects)***

$$(i). \quad : \quad A \equiv [l_i : B_i\,^{i\in 1..n}] \;\Rightarrow\; (res\ (\Sigma, \iota_i \mapsto (A \Rightarrow B_i)\,^{i\in 1..n})\ [l_i = \iota_i\,^{i\in 1..n}]\ A)$$

$$(ii). \quad : \quad A \equiv [l_i : B_i\,^{i\in 1..n}] \;\wedge\; (type\ [l_i = \varsigma(x_i)b_i\,^{i\in 1..n}]\ A) \;\wedge\; \Gamma, closed(x_i) \vdash (wrap\ b_i\ c_i)\,^{i\in 1..n} \;\wedge\; (comp\ \Sigma\ s) \;\Rightarrow\; (comp\ (\Sigma, \iota_i \mapsto (A \Rightarrow B_i)\,^{i\in 1..n})\ (s, \iota_i \mapsto \lambda x_i.c_i\,^{i\in 1..n}))$$

Proof. (i). By the rule (t_res).
(ii). By induction on the object type A. □

Lemma 6.11 ***(Select)***

$$(i). \quad : \quad (comp\ \Sigma\ s) \;\wedge\; s(\iota_i) = \lambda x.c \;\Rightarrow\; \Gamma, x \mapsto \Sigma_1(\iota_i) \vdash (type_{body}\ \Sigma\ c\ \Sigma_2(\iota_i))$$

Proof. By the rule (t_comp). □

Lemma 6.12 ***(Update)***

$$(i). \quad : \quad \Gamma, x \mapsto A \vdash (type\ b\ B) \;\wedge\; \Gamma, closed(x) \vdash (wrap\ b\ c) \;\wedge\; (\forall x, w, C.\ (x \mapsto w, x \mapsto C \in \Gamma) \Rightarrow \Gamma \vdash (res\ \Sigma\ w\ C)) \Rightarrow \Gamma, x \mapsto A \vdash (type_{body}\ \Sigma\ c\ B)$$

$$(ii). \quad : \quad \Gamma \vdash (comp\ \Sigma\ s) \;\wedge\; \Gamma, x \mapsto \Sigma_1(i) \vdash (type_{body}\ \Sigma\ c\ \Sigma_2(i)) \;\Rightarrow\; \Gamma \vdash (comp\ \Sigma\ (s.\iota_i \leftarrow \lambda x.c))$$

Proof. (i). By lemma 6.9.(iv).
(ii). By the rule (t_comp) and point (i). □

Lemma 6.13 ***(Clone)***

$$(i). \quad : \quad (res\ \Sigma\ [l_i = \iota_i\,^{i\in 1..n}]\ A) \;\Rightarrow\; (res\ (\Sigma, \iota'_i \mapsto \Sigma(\iota_i)\,^{i\in 1..n})\ [l_i = \iota'_i\,^{i\in 1..n}]\ A)$$

$$(ii). \quad : \quad (comp\ \Sigma\ s) \;\Rightarrow\; (comp\ (\Sigma, \iota'_i \mapsto \Sigma(\iota_i)\,^{i\in 1..n})\ (s, \iota'_i \mapsto s(\iota_i)\,^{i\in 1..n}))$$

Proof. (i). By induction on the result $[l_i = \iota_i\,^{i\in 1..n}]$.
(ii). By induction on the store type fragment $\iota'_i \mapsto \Sigma(\iota_i)\,^{i\in 1..n}$. □

We can now state and prove the Subject Reduction theorem for the full **imp**$_\varsigma$. Given the store type Σ, the coherence between types and results associated to the same variable in the context Γ is captured by:

$$\forall x, w, C.\ x \mapsto w,\ x \mapsto C \in \Gamma \;\Rightarrow\; \Gamma \vdash (res\ \Sigma\ w\ C)$$

It is worth noticing that the statement of the Subject Reduction is more involved with respect to the coinductive setting, thus leading to a longer and more difficult proof.

Theorem 6.10 ***(Subject Reduction, inductive setting, full impς)***
Let Γ be a well-formed context. Then:

$$\begin{array}{l}\Gamma \vdash (type\ a\ A)\ \wedge\ \Gamma \vdash (eval\ s\ a\ t\ v)\ \wedge\ \Gamma \vdash (comp\ \Sigma\ s)\ \wedge \\ (\forall x, w, C.\ (x \mapsto w, x \mapsto C \in \Gamma) \Rightarrow \Gamma \vdash (res\ \Sigma\ w\ C))\ \Rightarrow \\ \exists A^+ : TType, \Sigma^+ : SType. \\ \Gamma \vdash (res\ \Sigma^+\ v\ A^+)\ \wedge\ \Gamma \vdash (ext\ \Sigma^+\ \Sigma)\ \wedge\ \Gamma \vdash (comp\ \Sigma^+\ t)\ \wedge\ \Gamma \vdash (sub\ A^+\ A)\end{array}$$

Proof. By structural induction on the derivation of $\Gamma \vdash (eval\ s\ a\ t\ v)$. The rules *e_call* and *e_bind* require a *mutual* structural induction argument, namely a stronger induction schema valid also for the predicate $eval_{body}$, which is the counterpart of *eval* for closures. We present the proof by omitting the context Γ, that is, $\mathcal{J}$ stands for $\Gamma \vdash \mathcal{J}$.

(e_var). By hypothesis $(type\ x\ A)$ and:

$$\text{(e\_var)}\ \frac{x \mapsto v \in \Gamma}{(eval\ s\ x\ s\ v)}$$

From lemma 6.4.(var), there exists B such that $x \mapsto B \in \Gamma$ and $(sub\ B\ A)$. Choose $A^+ := B$ and $\Sigma^+ := \Sigma$.

Since $x \mapsto v \in \Gamma$, by the 4th hypothesis of the theorem we derive $(res\ \Sigma^+\ v\ A^+)$. We have $(ext\ \Sigma^+\ \Sigma^+)$ by lemma 6.9.(i) and $(comp\ \Sigma^+\ s)$ by hypothesis, thus concluding.

(e_obj). By hypothesis $(type\ [l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}]\ A)$ and:

$$\text{(e\_obj)}\ \frac{\begin{array}{c}(closed(x_i)) \\ \vdots \\ (l_i, \iota_i\ distinct) \quad \iota_i \notin dom(\sigma) \quad (wrap\ b_i\ c_i) \quad \forall i \in 1..n\end{array}}{(eval\ s\ [l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}]\ (s, \iota_i \mapsto \lambda x.c_i\ ^{i\in 1..n})\ [l_i = \iota_i\ ^{i\in 1..n}])}$$

By lemma 6.4.(obj), there exists $[l_i : B_i\ ^{i\in 1..n}]$ such that:

$$(type\ [l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}]\ [l_i : B_i\ ^{i\in 1..n}]) \tag{6.5}$$

$$(sub\ [l_i : B_i\ ^{i\in 1..n}]\ A) \tag{6.6}$$

Choose $A^+ := [l_i : B_i\ ^{i\in 1..n}]$ and $\Sigma^+ := \Sigma, \iota_i \mapsto (A^+ \Rightarrow B_i)^{\,i\in 1..n}$.

We have $(res\ \Sigma^+\ [l_i = \iota_i\ ^{i\in 1..n}]\ A^+)$ by lemma 6.10.(i) and it is immediate that $(ext\ \Sigma^+\ \Sigma)$. Next, since $(comp\ \Sigma\ s)$ and 6.5, we apply the lemma 6.10.(ii), thus deriving $(comp\ \Sigma^+\ (s, \iota_i \mapsto \lambda x_i.c_i\ ^{i\in 1..n}))$. We finish by 6.6.

(e_call). By hypothesis $(type\ (a.l_j)\ A)$ and:

$$\text{(e\_call)}\ \frac{\begin{array}{c}(x \mapsto [l_i = \iota_i\ ^{i\in 1..n}]) \\ \vdots \\ (eval\ s\ a\ s'\ [l_i = \iota_i\ ^{i\in 1..n}]) \quad j \in 1..n \quad s'(\iota_j) = \lambda x.c \quad (eval_{body}\ s'\ c\ t\ w)\end{array}}{(eval\ s\ (a.l_j)\ t\ w)}$$

By lemma 6.4.(call), there exists $[l_j : B_j, \ldots]$ such that $(type\ a\ [l_j : B_j, \ldots])$ and $(sub\ B_j\ A)$. Since $(eval\ s\ a\ s'\ [l_i = \iota_i\ ^{i\in 1..n}])$, by inductive hypothesis there exist C, Σ' such that:

(a). ($res\ \Sigma'\ [l_i = \iota_i{}^{i\in 1..n}]\ C$);
(b). ($ext\ \Sigma'\ \Sigma$);
(c). ($comp\ \Sigma'\ s'$);
(d). ($sub\ C\ [l_j : B_j, \ldots]$).
From the rule (e_call), it is $j \in 1..n$, $s'(\iota_j) = \lambda x.c$ and:

$$\Gamma, x \mapsto [l_i = \iota_i{}^{i\in 1..n}] \vdash (eval_{body}\ s'\ c\ t\ w) \tag{6.7}$$

On the other hand, we have $C \equiv [l_j : B_j, \ldots]$ from (d), thus $\Sigma'(\iota_j) = (C \Rightarrow B_j)$ and so, by (c) and lemma 6.11.(i):

$$\Gamma, x \mapsto C \vdash (type_{body}\ \Sigma'\ c\ B_j) \tag{6.8}$$

We deduce $(\forall x, w, C.\ (x \mapsto w, x \mapsto C \in \Gamma) \Rightarrow \Gamma \vdash (res\ \Sigma'\ w\ C))$ from the fourth hypothesis of the theorem and lemma 6.9.(iii); therefore, since 6.7, 6.8 and (c), we can apply the mutual induction hypothesis, thus concluding there exist A^+, Σ^+ such that:
(e). ($res\ \Sigma^+\ w\ A^+$);
(f). ($ext\ \Sigma^+\ \Sigma'$);
(g). ($comp\ \Sigma^+\ t$);
(h). ($sub\ A^+\ B_j$).
We finish by transitivity of ext (lemma 6.9.(ii)) and transitivity of subtyping.

(e_over). By hypothesis ($type\ (a.l \leftarrow \varsigma(x)b)\ A$) and:

$$\text{(e\_over)}\ \frac{\begin{array}{c}(closed(x))\\ \vdots\\ (eval\ s\ a\ s'\ [l_i = \iota_i{}^{i\in 1..n}])\quad j \in 1..n\quad \iota_j \in dom(s')\quad (wrap\ b\ c)\end{array}}{(eval\ s\ (a.l \leftarrow \varsigma(x)b)\ (s'.\iota_j \leftarrow \lambda x.c)\ [l_i = \iota_i{}^{i\in 1..n}])}$$

By lemma 6.4.(over), there exists $[l_j : B_j, \ldots]$ such that ($type\ a\ [l_j : B_j, \ldots]$), ($sub\ [l_j : B_j, \ldots]\ A$) and $\Gamma, x \mapsto [l_j : B_j, \ldots] \vdash (type\ b\ B_j)$. Since $(eval\ s\ a\ s'\ [l_i = \iota_i{}^{i\in 1..n}])$, we can apply the inductive hypothesis, thus deducing there exist C, Σ' such that:
(a). ($res\ \Sigma'\ [l_i = \iota_i{}^{i\in 1..n}]\ C$);
(b). ($ext\ \Sigma'\ \Sigma$);
(c). ($comp\ \Sigma'\ s'$);
(d). ($sub\ C\ [l_j : B_j, \ldots]$), that is, $C \equiv [l_j : B_j, \ldots]$.
Choose $A^+ := C$ and $\Sigma^+ := \Sigma'$.

By lemma 6.4.(bd-weak) we obtain $\Gamma, x \mapsto C \vdash (type\ b\ B_j)$, that is, using (a) and $j \in 1..n$:

$$\Gamma, x \mapsto \Sigma_1^+(\iota_j) \vdash (type\ b\ \Sigma_2^+(\iota_j)) \tag{6.9}$$

We derive $(\forall x, w, C.\ (x \mapsto w, x \mapsto C \in \Gamma) \Rightarrow \Gamma \vdash (res\ \Sigma^+\ w\ C))$ from the fourth hypothesis of the theorem and lemma 6.9.(iii). Next, because of $\Gamma, closed(x) \vdash (wrap\ b\ c)$ and 6.9, by lemma 6.12.(i):

$$\Gamma, x \mapsto \Sigma_1^+(\iota_j) \vdash (type_{body}\ c\ \Sigma_2^+(\iota_j)) \tag{6.10}$$

Since (c) and 6.10, we apply the lemma 6.12.(ii), thus deriving ($comp\ \Sigma^+\ (s'.\iota_j \leftarrow \lambda x.c)$). We conclude by transitivity of subtyping.

(e_clone). By hypothesis $(type\ (clone\ a)\ A)$ and:

$$\text{(e\_clone)}\ \frac{(eval\ s\ a\ s'\ [l_i = \iota_i\,^{i\in 1..n}])\ \iota_i \in dom(s')\ (\iota'_i\ distinct)\ \iota'_i \notin dom(s')\ \forall i \in 1..n}{(eval\ s\ (clone\ a)\ (s', \iota'_i \mapsto s'(\iota_i)\,^{i\in 1..n})\ [l_i = \iota'_i\,^{i\in 1..n}])}$$

By lemma 6.4.(clone), there exists B such that $(type\ a\ B)$ and $(sub\ B\ A)$. Since $(eval\ s\ a\ s'\ [l_i = \iota_i\,^{i\in 1..n}])$, we can apply the inductive hypothesis, thus deducing there exist C, Σ' such that:
(a). $(res\ \Sigma'\ [l_i = \iota_i\,^{i\in 1..n}]\ C)$;
(b). $(ext\ \Sigma'\ \Sigma)$;
(c). $(comp\ \Sigma'\ s')$;
(d). $(sub\ C\ B)$.
Choose $A^+ := C$ and $\Sigma^+ := \Sigma', \iota'_i \mapsto \Sigma'(\iota_i)\,^{i\in 1..n}$.

We deduce $(ext\ \Sigma^+\ \Sigma)$ by transitivity of the ext relation; similarly $(sub\ A^+\ A)$ by transitivity of subtyping. Next we have $(res\ \Sigma^+\ [l_i = \iota'_i\,^{i\in 1..n}]\ A^+)$ from (a) and lemma 6.13.(i) and finally $(comp\ \Sigma^+\ (s', \iota'_i \mapsto s'(\iota_i)\,^{i\in 1..n}))$ using (c) and lemma 6.13.(ii).

(e_let). By hypothesis $(type\ (let\ x = a\ in\ b)\ A)$ and:

$$\text{(e\_let)}\ \frac{(eval\ s\ a\ s'\ v) \quad \begin{array}{c}(x \mapsto v)\\ \vdots\\ (eval\ s'\ b\ t\ w)\end{array}}{(eval\ s\ (let\ x = a\ in\ b)\ t\ w)}$$

By lemma 6.4.(let), there exist B, C such that $(type\ a\ C)$ and $\Gamma, x \mapsto C \vdash (type\ b\ B)$ and $(sub\ B\ A)$. Since $(eval\ s\ a\ s'\ v)$, by inductive hypothesis there exist D, Σ' such that:
(a). $(res\ \Sigma'\ v\ D)$;
(b). $(ext\ \Sigma'\ \Sigma)$;
(c). $(comp\ \Sigma'\ s')$;
(d). $(sub\ D\ C)$.
Since $\Gamma, x \mapsto C \vdash (type\ b\ B)$ and (b), we use lemma 6.4.(bd-weak) for deriving $\Gamma, x \mapsto D \vdash (type\ b\ B)$. Next we deduce $(\forall x, w, C.\ (x \mapsto w, x \mapsto C \in \Gamma) \Rightarrow \Gamma \vdash (res\ \Sigma'\ w\ C))$ from (a) and lemma 6.9.(iii). Because of $\Gamma, x \mapsto v \vdash (eval\ s'\ b\ t\ w)$, we apply again the induction hypothesis, thus obtaining E, Σ'' such that:
(e). $(res\ \Sigma''\ w\ E)$;
(f). $(ext\ \Sigma''\ \Sigma')$;
(g). $(comp\ \Sigma''\ t)$;
(h). $(sub\ E\ B)$.
Choose $A^+ := E$ and $\Sigma^+ := \Sigma''$.

We conclude by transitivity of ext and transitivity of subtyping. □

(e_ground). By hypothesis $(type_{body}\ \Sigma\ (ground\ a)\ A)$ and:

$$\text{(e\_ground)}\ \frac{(eval\ s\ a\ t\ v)}{(eval_{body}\ s\ (ground\ a)\ t\ v)}$$

The assertion $(type_{body}\ \Sigma\ (ground\ a)\ A)$ has to be derived by the rule (t_ground), namely from $(type\ a\ A)$; therefore, by induction, there exist A^+, Σ^+ s.t. $(res\ \Sigma^+\ v\ A^+)$, $(ext\ \Sigma^+\ \Sigma)$, $(comp\ \Sigma^+\ t)$ and $(sub\ A^+\ A)$.

(e_bind). By hypothesis $(type_{body}\ \Sigma\ (bind\ v\ \lambda y.c)\ A)$ and:

$$(\text{e\_bind})\ \frac{\begin{array}{c}(y \mapsto v)\\ \vdots\\ (eval_{body}\ s\ c\ t\ w)\end{array}}{(eval_{body}\ s\ (bind\ v\ \lambda y.c)\ t\ w)}$$

The assertion $(type_{body}\ \Sigma\ (bind\ v\ \lambda y.c)\ A)$ has to be derived by the rule (t_bind), namely there exists B such that $(res\ \Sigma\ v\ B)$ and $\Gamma, y \mapsto B \vdash (type_{body}\ \Sigma\ c\ A)$. Therefore, by mutual induction, there exist A^+, Σ^+ such that $(res\ t\ w\ A^+)$, $(ext\ \Sigma^+\ \Sigma)$, $(comp\ \Sigma^+\ t)$ and $(sub\ A^+\ A)$. □

$$(\text{c\_obj})\ \frac{\begin{array}{c}(closed(x_i))\\ \vdots\\ (closed\ b_i) \quad \forall i \in 1..n\end{array}}{(closed\ [l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}])}$$

$$(\text{c\_call})\ \frac{(closed\ a)}{(closed\ (call\ a\ l))}$$

$$(\text{c\_over})\ \frac{\begin{array}{cc}(closed\ a) & \begin{array}{c}(closed(x))\\ \vdots\\ (closed\ b)\end{array}\end{array}}{(closed\ (over\ a\ l\ \varsigma(x)b))}$$

$$(\text{c\_clone})\ \frac{(closed\ a)}{(closed\ (clone\ a))}$$

$$(\text{c\_let})\ \frac{\begin{array}{cc}(closed\ a) & \begin{array}{c}(closed(x))\\ \vdots\\ (closed\ b)\end{array}\end{array}}{(closed\ (let\ a\ \lambda x.b))}$$

$$(\text{w\_ground})\ \frac{(closed\ b)}{(wrap\ b\ (ground\ b))}$$

$$(\text{w\_bind})\ \frac{\begin{array}{cc}\begin{array}{c}(closed(z))\\ \vdots\\ (wrap\ b\{z/y\}\ c\{z/y\})\end{array} & y \mapsto v \in \Gamma\end{array}}{(wrap\ b\ (bind\ v\ \lambda y.c))}\ (z\ fresh)$$

Figure 6.1: Natural Deduction formation of closures for **impς**

$$\text{(e\_var)}\ \frac{x \mapsto v \in \Gamma}{(eval\ s\ x\ s\ v)}$$

$$\text{(e\_obj)}\ \frac{\begin{array}{c}(closed(x_i))\\ \vdots\\ (l_i, \iota_i\ distinct) \quad \iota_i \notin dom(\sigma) \quad (wrap\ b_i\ c_i) \quad \forall i \in 1..n\end{array}}{(eval\ s\ [l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}]\ (s, \iota_i \mapsto \lambda x.c_i\ ^{i\in 1..n})\ [l_i = \iota_i\ ^{i\in 1..n}])}$$

$$\text{(e\_call)}\ \frac{\begin{array}{c}(x \mapsto [l_i = \iota_i\ ^{i\in 1..n}])\\ \vdots\\ (eval\ s\ a\ s'\ [l_i = \iota_i\ ^{i\in 1..n}]) \quad j \in 1..n \quad s'(\iota_j) = \lambda x.c \quad (eval_{body}\ s'\ c\ t\ w)\end{array}}{(eval\ s\ (a.l_j)\ t\ w)}$$

$$\text{(e\_over)}\ \frac{\begin{array}{c}(closed(x))\\ \vdots\\ (eval\ s\ a\ s'\ [l_i = \iota_i\ ^{i\in 1..n}]) \quad j \in 1..n \quad \iota_j \in dom(s') \quad (wrap\ b\ c)\end{array}}{(eval\ s\ (a.l \leftarrow \varsigma(x)b)\ (s'.\iota_j \leftarrow \lambda x.c)\ [l_i = \iota_i\ ^{i\in 1..n}])}$$

$$\text{(e\_clone)}\ \frac{(eval\ s\ a\ s'\ [l_i = \iota_i\ ^{i\in 1..n}]) \quad \iota_i \in dom(s') \quad (\iota'_i\ distinct) \quad \iota'_i \notin dom(s') \quad \forall i \in 1..n}{(eval\ s\ (clone\ a)\ (s', \iota'_i \mapsto s'(\iota_i)\ ^{i\in 1..n})\ [l_i = \iota'_i\ ^{i\in 1..n}])}$$

$$\text{(e\_let)}\ \frac{\begin{array}{c}(x \mapsto v)\\ \vdots\\ (eval\ s\ a\ s'\ v) \quad (eval\ s'\ b\ t\ w)\end{array}}{(eval\ s\ (let\ x = a\ in\ b)\ t\ w)}$$

$$\text{(e\_ground)}\ \frac{(eval\ s\ a\ t\ v)}{(eval_{body}\ s\ (ground\ a)\ t\ v)}$$

$$\text{(e\_bind)}\ \frac{\begin{array}{c}(y \mapsto v)\\ \vdots\\ (eval_{body}\ s\ c\ t\ w)\end{array}}{(eval_{body}\ s\ (bind\ v\ \lambda y.c)\ t\ w)}$$

Figure 6.2: Natural Deduction Operational Semantics for **imp**ς

$$\text{(wt\_obj)}\ \frac{(wt\ B_i) \quad (l_i\ distinct) \quad \forall i \in 1..n}{(wt\ [l_i : B_i{}^{i\in 1..n}])}$$

$$\text{(sub\_refl)}\ \frac{(wt\ A)}{(sub\ A\ A)} \qquad \text{(sub\_trans)}\ \frac{(sub\ A\ B) \quad (sub\ B\ C)}{(sub\ A\ C)}$$

$$\text{(sub\_obj)}\ \frac{(wt\ B_i) \quad (l_i\ distinct) \quad \forall i \in 1..n+m}{(sub\ [l_i : B_i{}^{i\in 1..n+m}]\ [l_i : B_i{}^{i\in 1..n}])}$$

$$\text{(t\_sub)}\ \frac{(type\ a\ A) \quad (sub\ A\ B)}{(type\ a\ B)}$$

$$\text{(t\_var)}\ \frac{(wt\ A) \quad x \mapsto A \in \Gamma}{(type\ x\ A)}$$

$$\text{(t\_obj)}\ \frac{\begin{array}{c}(x_i \mapsto [l_i : B_i{}^{i\in 1..n}]) \\ \vdots \\ (wt\ [l_i : B_i{}^{i\in 1..n}]) \qquad (type\ b_i\ B_i) \qquad \forall i \in 1..n\end{array}}{(type\ [l_i = \varsigma(x_i)b_i{}^{i\in 1..n}]\ [l_i : B_i{}^{i\in 1..n}])}$$

$$\text{(t\_call)}\ \frac{(type\ a\ [l_i : B_i{}^{i\in 1..n}]) \quad j \in 1..n}{(type\ (a.l_j)\ B_j)}$$

$$\text{(t\_over)}\ \frac{\begin{array}{c}(x \mapsto [l_i : B_i{}^{i\in 1..n}]) \\ \vdots \\ (type\ a\ [l_i : B_i{}^{i\in 1..n}]) \quad j \in 1..n \qquad (type\ b\ B_j)\end{array}}{(type\ (a.l \leftarrow \varsigma(x)b)\ [l_i : B_i{}^{i\in 1..n}])}$$

$$\text{(t\_clone)}\ \frac{(type\ a\ [l_i : B_i{}^{i\in 1..n}])}{(type\ (clone\ a)\ [l_i : B_i{}^{i\in 1..n}])}$$

$$\text{(t\_let)}\ \frac{\begin{array}{c}(x \mapsto A) \\ \vdots \\ (type\ a\ A) \quad (type\ b\ B)\end{array}}{(type\ (let\ x = a\ in\ b)\ B)}$$

Figure 6.3: Natural Deduction Term Typing for **impς**

$$\text{(t\_cores)}\ \frac{\begin{array}{c}(x \mapsto [l_i : B_i\,^{i\in 1..n}]),\ (\mathcal{J}) \\ \vdots \\ \begin{array}{ccc}(wt\ [l_i : B_i\,^{i\in 1..n}]) & & (distinct\ \iota_i) \\ s(\iota_i) = \lambda x_i.c_i & (cotype_{body}\ s\ c_i\ B_i) & \iota_i \in dom(s) \quad \forall i \in 1..n\end{array}\end{array}}{(cores\ s\ [l_i = \iota_i\,^{i\in 1..n}]\ [l_i : B_i\,^{i\in 1..n}]) \equiv \mathcal{J}}$$

$$\text{(t\_coground)}\ \frac{(type\ b\ A)}{(cotype_{body}\ s\ (ground\ b)\ A)}$$

$$\text{(t\_cobind)}\ \frac{\begin{array}{c}(y \mapsto A) \\ \vdots \\ (cores\ s\ v\ A) \quad (cotype_{body}\ s\ c\ B)\end{array}}{(cotype_{body}\ s\ (bind\ v\ \lambda y.c)\ B)}$$

Figure 6.4: Coinductive Natural Deduction Result Typing for **imp**$_\varsigma$

$$\text{(t\_res)}\ \frac{\begin{array}{c}(distinct\ \iota_i) \quad \iota_i \in dom(\Sigma) \quad \forall i \in 1..n \\ \Sigma_1(\iota_i) \equiv [l_i : \Sigma_2(\iota_i)\,^{i\in 1..n}] \quad (wt\ [l_i : \Sigma_2(\iota_i)\,^{i\in 1..n}])\end{array}}{(res\ \Sigma\ [l_i = \iota_i\,^{i\in 1..n}]\ [l_i : \Sigma_2(\iota_i)\,^{i\in 1..n}])}$$

$$\text{(t\_ext)}\ \frac{(dom\ \Sigma) \subseteq (dom\ \Sigma') \qquad \forall \iota_i \in (dom\ \Sigma).\ \Sigma'(\iota_i) = \Sigma(\iota_i)}{(ext\ \Sigma'\ \Sigma)}$$

$$\text{(t\_ground)}\ \frac{(type\ b\ A)}{(type_{body}\ \Sigma\ (ground\ b)\ A)}$$

$$\text{(t\_bind)}\ \frac{\begin{array}{c}(y \mapsto A) \\ \vdots \\ (res\ \Sigma\ v\ A) \quad (type_{body}\ \Sigma\ c\ B)\end{array}}{(type_{body}\ \Sigma\ (bind\ v\ \lambda y.c)\ B)}$$

$$\text{(t\_comp)}\ \frac{\begin{array}{c}(x_i \mapsto \Sigma_1(\iota_i)) \\ \vdots \\ (dom\ s) \subseteq (dom\ \Sigma) \quad s(\iota_i) = \lambda x_i.c_i \quad (type_{body}\ c_i\ \Sigma_2(\iota_i)) \quad \forall \iota_i \in dom(s)\end{array}}{(comp\ \Sigma\ s)}$$

Figure 6.5: Inductive Natural Deduction Result Typing for **imp**$_\varsigma$

Chapter 7

A Type Sound formalization of impς in Coq

This chapter presents the formalization of Abadi-Cardelli's **imp**ς-calculus in Coq and the development of the corresponding metatheory, culminating in the Type Soundness. We follow the reformulation of **imp**ς stated in the previous chapter, adopting the higher-order abstract syntax encoding approach (HOAS).

7.1 Encoding methodology

We have given in the previous chapter a reformulation of **imp**ς, "with paper and pencil", better suited for an encoding in Coq. Its formalization in the specification language of the proof assistant remains a complex task, although much simpler than the original system of chapter 5. Anyway, we have to face many subtle details which are left "implicit" on the paper, typically described by the use of natural or mathematical language. An immediate advantage is that we can use well-known encoding methodologies (see e.g. [PE88, HHP93, DFH95, Sca02]), therefore the adequacy of the encoding with respect to the presentation of the previous chapter can be proved easily.

One source of complications is the presence of binders in the **imp**ς-calculus. The first one is ς, which is used in every right-hand part of components of objects, i.e. methods: in the expression $\varsigma(x)b$, ς binds the free variable x in b. The second binder is the *let* construct: the expression $let\ x = a\ in\ b$ binds the free variable x in b.

Binders are known to be difficult to deal with; we would the metalanguage to take care of all the burden of α-conversion, substitutions, variable scope and so on. In recent years, many approaches have been proposed for dealing with binders, which essentially differ from the expressive power of the underlying metalanguage [Hir97, Mic97, MAC01, HMS01b]. Among them, encoding methodologies largely used in logical frameworks are *first-order* and *higher-order* abstract syntax.

Using the first-order approach (FOAS) one can choose between *de Bruijn indexes* and *explicit names*, but in both cases the managing of binders is problematic. The treatment of de Bruijn indexes is a daunting task and is difficult to understand for humans: it produces a surcharge that can be greater than the management of all the other aspects (e.g. the 75% of the lemmas proved in [Hir97] concern technical manipulations of the indexes). On the other hand, the use of explicit names charges the user with the burden of encoding the

mechanism of α-*equivalence*: this effort represents a non-trivial task from the perspective of the computer aided proof development, because α-equivalence is usually given on the paper through a short definition in natural language.

Alternatively, the higher-order approach (HOAS) allows to express in a uniform and algebraic way binding operators over variables; therefore it is useful to express context dependent constraints, and provides a standard format for treating α-conversion and scope disciplines.

Encoding choices. Among the many possibilities, we have chosen the *second-order abstract syntax*, called also "weak HOAS" [DFH95, Mic97, HMS01a, HMS01b]. In this approach, binding operators are represented by constructors of higher order type. The main difference with respect to the full HOAS is that abstractions range over unstructured (i.e., non inductive) sets of *abstract variables*. In this way, α-conversion is automatically provided by the metalanguage, while substitution of terms for variables is not. This fits perfectly the needs for the encoding of **imp**ς, since the language is taken up-to α-equivalence, and substitution of terms for variables is not needed in the statement of the semantics. Moreover the weak HOAS is compatible with inductive datatypes, still avoiding the arising of *exotic terms*; therefore, we can establish easily the adequacy of the encoding. An issue of HOAS is related to the difficulty of reasoning by induction and using recursion over contexts, since they are rendered as functional terms; one also looses the possibility of handling and proving properties over the mechanisms delegated to the metalanguage.

Our choice of using HOAS (and not, for example, a first-order approach) is also motivated by the fact that we would like to reuse our formalization for reasoning about equivalences of programs of **imp**ς, which is α-equivalence: that is, two terms, or objects, behave in the same way.

Theory of Contexts. When we have to develop metatheoretical results in a HOAS setting, the expressive power of $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$, the (co)inductive type theory underlying Coq, may be not enough. In [HMS01b, Sca02] a general methodology for reasoning on systems in HOAS is presented. The gist is to extend the framework with a set of axioms, called the *Theory of Contexts*, capturing some basic and natural properties of *names* and *term contexts*. These axioms allow for a smooth handling of schemata in HOAS, with a very low mathematical and logical overhead. The present work can thus be seen as an extensive case study of the application of that theory, for the first time, to an imperative object-oriented calculus.

7.2 Formalization in Coq

The encoding of **imp**ς in Coq follows naturally from the rephrasing of the previous chapter. It is worth saying that we will not address formally the adequacy of this second translation of the calculus, because we adopt encoding techniques which are common practice. An immediate exemplification is the use of inductive structures for representing finite collections of components: this is the case of objects (collections of pairs), types and results. These and other encodings are detailed in the progress of the current section.

Variables, metavariables and names. A crucial aspect of the formalization is the use of the metavariables of Coq for representing the variables of the object calculus; that is, for distinguishing *names* from other names. This is particularly delicate when considering hypothetical premises of associations between variables and other entities (as results or types), a situation peculiar of our natural deduction approach.

When we reason on hypothetical premises, we gain for free in the proof assistant the generation of new metavariables, but we do not know a priori whether two different metavariables x and y denote *different* names. Moreover we do not know whether a given metavariable occurs free or not in a term. In fact, reasoning about names in HOAS is not fully supported in current logical frameworks, Coq in our case. In order to get the extra expressive power we need, we extend the framework with the Theory of Contexts, which allow for a smooth manipulation of these notions related to names and variables.

Our management of names causes the splitting of the universe of variables we use, say Var, into two disjoint sub-universes. We use actually metavariables with two different flavours: first we need to use names for implementing environments; second we have to dispose of placeholders for reasoning about occurrences of variables in (higher-order) terms. So doing, we have to force metavariables in hypothetical premises to satisfy additional and different conditions, depending on the particular concept being defined.

This distinction reflects the difference between theory and metatheory of $\mathbf{imp}_{\varsigma}$. Theory and metatheory actually require the formalization of different notions and judgments, with the result of mixing theoretical and metatheoretical information in hypothetical premises of the rules, as will be shown later in the section.

Syntax.

We start presenting the encoding of terms and types of $\mathbf{imp}_{\varsigma}$.

Terms. The signature of the weak HOAS-based encoding of the syntax is the following:

```
Parameter Var : Set.
Definition Lab := nat.

Inductive Term : Set := var  : Var -> Term
                      | obj  : Object -> Term
                      | call : Term -> Lab -> Term
                      | over : Term -> Lab -> (Var -> Term) -> Term
                      | clone: Term -> Term
                      | let  : Term -> (Var -> Term) -> Term
with Object : Set := obj_nil : Object
                   | obj_cons: Lab -> (Var -> Term) -> Object -> Object.
Coercion var : Var >-> Term.
```

Notice first that we use a separate type `Var` for variables (weak HOAS setting): the only terms which can inhabit this type are the variables of the metalanguage, that represent directly the variables of the object language. The alternative would be to define `Var` by an inductive type, but there is no reason to bring in unnecessary assumptions, namely the induction and recursion principles: these unwanted principles would introduce inconsistencies with the Theory of the Contexts [Sca02]. In fact, if Var were inductive, we

could define *exotic terms* using the `Case` construct of Coq: exotic terms are terms which do not correspond to any expression of **imp**ς, and therefore they have to be ruled out by extra "well-formedness" judgments, thus complicating the whole encoding.

We make the naturals $I\!N$ playing the role of the type `Lab` of labels, i.e. names of methods.

Terms are formalized by an inductive type, whose constructors correspond to the different constructs of the calculus. In turn, we have to choose an inductive data structure for representing objects, i.e. finite collections of methods. We prefer to introduce specific constructors for objects, isomorphic to lists; the alternative would be to use directly polymorphic lists, for example defining objects by:

```
obj :  (list (Lab * (Var -> Term))) -> Term
```

A previous attempt has shown that this choice would not allow for defining by recursion on structure of terms some fundamental functions, essential for the rest of the formalization. Using lists, the specification of these functions would be "unguarded", while they are feasible in the definition above.

Therefore we define terms (`Term`) and objects (`Object`) by mutual induction: `obj_cons`, `over` and `let` are higher-order constructors, since they take a functional term as argument. Such a term, `Var->Term`, represents exactly the *capture-avoiding contexts* of the **imp**ς-calculus. This technique allows to inherit the α-equivalence on terms from the metalanguage and also to have available induction principles for terms: namely, we dispose of induction proof principles and we can define functions by first-order recursion or case analysis on the syntax of terms.

The inhabitants of the functional type `Var->Term` stand for *methods* in the case of the constructors `obj_cons` and `over`, for *local definitions* in the case of the `let` constructor. For instance, the two objects $[m = \varsigma(x)x]$ and $[m = \varsigma(y)y]$ can be represented by:

```
(obj (obj_cons m [x:Var]x obj_nil))
(obj (obj_cons m [y:Var]y obj_nil))
```

which are α-equivalent terms.

It is also worth noticing that, in local declarations $let\ x = a\ in\ b$, we have to take care that the variable x is bound in b. Thus, for example, we represent $let\ x = a\ in\ x$ by:

```
(let a [x:Var]x)
```

Finally note that declaring the constructor `var` as a coercion permits to inject implicitly terms from `Var` into `Term`, in such a way that the constructor `var` can be omitted from terms.

Types. The types of **imp**ς are encoded by the following structure:

```
Inductive TType : Set := mktype: (list (Lab * TType)) -> TType.
```

Object types are inductive concrete sets with only one constructor, that builds object types from lists of pairs. This solution is slightly different with respect to the one we have adopted for the syntax of the objects, but we do not lose the complete correspondence between the inductive structures of objects and object types.

Polymorphic lists of Coq are quite well-suited for encoding entities whose nature is intrinsically inductive, as finite collections. Lists and basic properties about them, defined in the library `PolyList`, are used systematically in the following.

Dynamic Semantics.

We introduce first all the entities required by the operational semantics corresponding to the abstract formulation of **imp**ς. Notice that, when one uses Coq, all the semantic structures introduced in the previous chapter have to be fully implemented and managed.

Entities. A *store location* is a natural number pointing to the *global store*, a finite sequence of *method closures*, implemented by a list:

```
Definition Loc := nat.
Definition Store : Set := (list Closure).
Definition size : Store -> nat := [s:Store] (length s).
Definition alloc : Store -> (list Closure) -> Store :=
           [s:Store; cl:(list Closure)] (app s cl).
Fixpoint update [n:nat] : Closure -> Store -> Store :=
         [c:Closure; s:Store]
         Cases n s of O     (cons hd tl) => (cons c tl)
                   |  (S m) (cons hd tl) => (cons hd (update m c tl))
                   |  _     nil => s
         end.
```

We have encoded the store by means of a linear structure as simplest as possible: its dimension can be recovered through the function *size* —which coincides with the built-in function *lenght* on polymorphic lists— and it can be accessed sequentially using the pointers from 0 to $size(s)-1$. The store can be extended with a concatenation of a list of closures —through the built-in *app* (append) function. A store can be modified by *update*, a total function —as required by Coq— modeled by recursion. Its definition is delicate, because it performs intrinsically partial operations: *update* is designed in such a way that if the pointer is dangling (i.e. it is greater or equal to the size of the store), the store is left unchanged. Such a behavior is easily recognized as "abnormal" during the formal proofs, and thus can be managed coherently.

We have explained in the previous chapter that our method closures are different w.r.t. the original Abadi-Cardelli's ones: we do not require to store in the memory methods and stacks, but only *methods*, transformed in such a way that they do not contain free variables anymore. Thus, closures are bodies abstracted with respect to the `self` variable:

```
Definition Closure : Set := (Var -> Body).
Inductive Body : Set := ground : Term -> Body
                      | bind   : Res -> (Var -> Body) -> Body.
```

Notice that closure bodies are an inductive datatype with an higher-order constructor similar to *let*. We have remarked that our design of closures is consequence of the fact that, adhering to the natural deduction semantics (NDS) approach, we do not implement explicitly the environment. This choice simplifies the statement of the judgments and the

development of the formal proofs and permits to get "optimized" closures, because only variables which are really free in the body need to be bound in the closure.

Therefore, one of the crucial aspects of the NDS approach is the use of hypothetical premises, namely environment fragments corresponding to the explicit Abadi-Cardelli's stack S. The environmental information of the stack is represented as a function, associating a result to each (declared) variable. Results are lists of pairs built of method names and pointers to the corresponding closures in the store:

```
Parameter stack : Var -> Res.
Definition Res : Set := (list (Lab * Loc)).
```

This map is never defined effectively: `(stack x v)` corresponds to $x \mapsto v$, which is discharged from the proof environment but never proved as a judgment. Correspondingly, assumptions regarding `stack` will be discharged in the rules in order to associate results to variables.

Auxiliary functions. At this point we have to implement a certain package of functions for manipulating the explicit structures we have introduced so far. We need functions for projecting single lists from lists of pairs ($proj\_lab\_obj, proj\_lab\_res, proj\_meth\_obj, proj\_loc\_res$), for generating new results from objects and results (new_res_obj, new_res_res), for visiting and managing the store ($store\_nth, store\_to\_list, loc\_in\_res$) and for comparing labels ($eq\_lab, distinct$):

```
Fixpoint proj_lab_obj [ml:Object] : (list Lab) :=
         Cases ml of obj_nil => (nil Lab)
                  |  (obj_cons l m nl) => (cons l (proj_lab_obj nl))
         end.
Fixpoint proj_lab_res [rl:Res] : (list Lab) :=
         Cases rl of nil => (nil Lab)
                  |  (cons (pair l x) sl) => (cons l (proj_lab_res sl))
         end.
Fixpoint proj_meth_obj [ml:Object] : (list (Var->Term)) :=
         Cases ml of obj_nil => (nil (Var->Term))
                  |  (obj_cons l m nl) => (cons m (proj_meth_obj nl))
         end.
Fixpoint proj_loc_res [rl:Res] : (list Loc) :=
         Cases rl of nil => (nil Loc)
                  |  (cons (pair l x) sl) => (cons x (proj_loc_res sl))
         end.
Fixpoint new_res_obj [ml:Object] : nat -> Res := [n:nat]
         Cases ml of obj_nil => (nil (Lab*Loc))
                  |  (obj_cons l m nl) => (cons (l,n)
                                                (new_res_obj nl (S n)))
         end.
Fixpoint new_res_res [rl:Res] : nat -> Res := [n:nat]
         Cases rl of nil => (nil (Lab*Loc))
                  | (cons (pair l x) sl) => (cons (l,n)
                                                  (new_res_res sl (S n)))
```

```
        end.
Definition store_nth [n:Loc; s:Store] : Closure := (nth n s void_closure).
Fixpoint store_to_list [il:(list Loc)] : Store -> (list Closure) :=
        [s:Store]
        Cases il of nil => (nil Closure)
                 |  (cons i jl) => (cons (store_nth i s)
                                         (store_to_list jl s)) end.
Fixpoint loc_in_res [v:Res] : Lab -> Store -> Loc := [l:Lab; s:Store]
        Cases v of nil => (size s)
                |  (cons (pair k i) w) =>
        Cases (eq_lab k l) of true => i
                            | false => (loc_in_res w l s) end.
Fixpoint eq_lab [m,n:Lab] : bool :=
        Cases m n of O O => true
                  |  (S p) (S q) => (eq_lab p q)
                  |  (S p) O => false
                  |  O (S q) => false end.
Fixpoint distinct [ll:(list Lab)] : Prop :=
        Cases ll of nil => True
                 |  (cons l kl) => ~(In l kl) /\ (distinct kl) end.
```

The function *loc_to_res*, similarly to *update* one, presented above, performs an intrinsically partial operation, but has to be totally defined in Coq. It actually searches for the pointer associated to a certain label, and so it is designed in such a way that if a label has not been found, it returns a dangling pointer. Notice that such behavior is simply seen as "abnormal" during the formal proofs. The definitions of the other functions are straightforward.

Extra notions and judgments. We recall the two following extra judgments, whose introduction is required for dealing with method closures:

$$\begin{array}{lcl} closed & \subseteq & Term \\ wrap & \subseteq & Term \times Body \end{array}$$

We prefer to formalize the notion *closed* by means of a function, in order to simplify the statement of the operational semantics and the proofs in Coq. The intended behavior of this function, defined by mutual recursion on the structure of terms and objects, is to propagate in depth the notion *closed*, thus reducing a predicate `(closed a)` into a predicate about simpler terms:

```
Parameter dummy : Var -> Prop.
Fixpoint closed [t:Term] : Prop := Cases t of
        (var x) => (dummy x)
    |   (obj ml) => (closed_obj ml)
    |   (over a l m) => (closed a) /\
                        ((x:Var)(dummy x)->(closed (m x)))
    |   (call a l) => (closed a)
    |   (clone a) => (closed a)
```

```
    |    (lEt a b) => (closed a) /\
                       ((x:Var)(dummy x)->(closed (b x))) end
with closed_obj [ml:Object] : Prop := Cases ml of
         (obj_nil) => True
    |    (obj_cons l m nl) => (closed_obj nl) /\
                              ((x:Var)(dummy x)->(closed (m x))) end.
```

In the translation, we use locally quantified variables as placeholders for bound variables; thus they have not to be considered as "free" variables. We mark them by an auxiliary predicate `dummy`, where dummy variables are considered "closed". The proposition resulting from the simplification of a `(closed a)` goal is easily dealt with using the tactics provided by Coq. The formalization of the function *closed* is paradigmatic about the use of hypothetical premises in an higher-order specification language.

The relation *wrap*, describing the construction of the method closures, requires to introduce the additional functions $notin : Var \times Term \rightarrow Prop$ and $fresh : Var \times (list\ Var) \rightarrow Prop$, which capture the "freshness" of a variable in a term and w.r.t. a list of variables, respectively. The former is defined by mutual induction, the latter by induction:

```
Fixpoint notin [y:Var; t:Term] : Prop := Cases t of
         (var x) => ~(y=x)
    |    (obj ml) => (notin_obj y ml)
    |    (over a l m) => (notin y a) /\
                         ((x:Var) ~(y=x) -> (notin y (m x)))
    |    (call a l) => (notin y a)
    |    (clone a) => (notin y a)
    |    (lEt a b) => (notin y a) /\
                       ((x:Var) ~(y=x) -> (notin y (b x))) end
with notin_obj [y:Var; ml:Object] : Prop := Cases ml of
         obj_nil => True
    |    (obj_cons l m nl) => (notin_obj y nl) /\
                              ((x:Var) ~(y=x) -> (notin y (m x))) end.
Fixpoint fresh [x:Var; l:(list Var)] : Prop := Cases l of
         nil => True
   |     (cons y yl) => ~(x=y) /\ (fresh x yl) end.
```

The function $fresh$ is useful for developing the *theory* of **imp**ς, namely for verifying the construction of closures. So we are mixing theoretical and metatheoretical notions in hypothetical premises. Finally, the judgment *wrap* is formalized via an inductive predicate, that uses all the concepts defined so far:

```
Inductive wrap : Term -> Body -> Prop:=
      w_ground : (b:Term)
                 (closed b) -> (wrap b (ground b))
   |  w_bind   : (b:Var->Term; c:Var->Body; y:Var; v:Res; xl:Varlist)
                 ((z:Var) (dummy z) /\ (fresh z xl) ->
                          (wrap (b z) (c z))) ->
                 (stack y) = (v) ->
```

```
((z:Var) ~(y=z) -> (notin y (b z))) ->
(wrap (b y) (bind v c)).
```

No transformation of a method body is necessary if it does not contain free variables. A different management is required in the case free variables appear in a method: the closure is built picking out a suitable result in the context and reserving it in order it can be bound to the right variable at the moment of the evaluation. It is worth noticing that, in the rule `w_bind`, the premise `((z:Var) (y=z) -> (notin y (b z)))` ensures that `b` is a "good context" for `y`, that is, `y` does not occur free in `b`. Thus, the replacement $b\{z/y\}$ of rule (*w_bind*) of Figure 6.1 can be implemented simply as the application `(b z)`, where `z` is a local (i.e., fresh) variable. The predicate *wrap* is used in the main reduction predicate at the moment of reasoning about method closures loaded in the store.

Term reduction judgment. We have explained in the previous chapter that our reformulation of **imp**ς describes the reduction by means of two mutually defined relations:

```
Mutual Inductive eval : Store -> Term -> Store -> Res -> Prop := ...
       with eval_body : Store -> Body -> Store -> Res -> Prop := ...
```

We formalize and discuss the constructors of the two predicates, that correspond to the rules of Figure 6.2. We remark that the rules for variables and local declarations enlighten how the proof environment is used to represent the stack.

The semantics of *variables* (*e_var*) does not require any particular explanation; evaluating x produces the result associated to x, through the function *stack*, in the current context:

```
e_var  : (s:Store) (x:Var) (v:Res)
         (stack x) = (v) -> (eval s x s v)
```

The rule for the *let* construct shows how to encode in Coq hypothetical premises; the completely detailed version of the rule (*e_let*) takes into account that b is an higher-order term:

```
e_let  : (s,s',t:Store) (a:Term) (b:Var->Term) (v,w:Res)
         (eval s a s' v) ->
         ((x:Var) (stack x) = (v) ->
                  (eval s' (b x) t w)) ->
         (eval s (let a b) t w)
```

The formalization of the semantics of *objects* looks very different with respect to the corresponding rule (*e_obj*), because we have to express that objects are a finite, possibly empty, collection of methods. Thus we carry out a preliminary transformation of the method list, for checking, by recursion, that methods have distinct names, and also for describing, through the *wrap* predicate, the relationship between the method bodies and their corresponding closures. This is performed by the following recursive function:

```
Fixpoint scan [ml:(list (Var->Term)); cl:(list Closure)] :
         Varlist -> Prop -> Prop := [xl:Varlist; P:Prop]
         Cases ml cl of nil nil => P
                     |  (cons m nl) (cons c dl) =>
```

```
                    (scan nl dl xl P) /\
                    ((x:Var) (dummy x) /\ (fresh x xl) ->
                             (wrap (m x) (c x)))
                |   _ _ => False
        end.
```

The two functions *alloc* and *new_res_obj* build a new store and a new result. The function *alloc* simply appends the new list of closures to the old store. The function *new_res_obj* produces a new result, collecting the methods names of a given object and pairing them to new pointers to the store. The process of generating pointers follows the usual order on natural numbers: at each step, i.e. for every method, we increment a counter pointing to the next free location. Thus new addresses are created only in correspondence of the effective loading of closures in the memory, so dangling references are never generated. This approach produces a possible implementation of the rule (*e_obj*), which is perfectly coherent with it:

```
e_obj  : (s,t:Store) (ml:Object) (cl:(list Closure))
         (v:Res) (xl:Varlist)
         (scan (proj_meth_obj (ml)) (cl) (xl)
               (distinct (proj_lab_obj ml))) ->
         (t = (alloc s cl)) ->
         (v = (new_res_obj ml (size s))) ->
         (eval s (obj ml) t v)
```

Also *method update* needs the *wrap* notion, because the overrider method has to be transformed in a closure. We use here the function *loc_in_res*, searching for the pointer associated to a certain method name in a result. Notice again that if a method name is found, then the associated pointer is not dangling; alternatively, if a certain method name does not occur in a result, a dangling reference is returned, pointing to the next available address. The encoding of the (*e_over*) rule requires also to use the *update* function:

```
e_over : (s,s',t:Store) (a:Term) (m:Var->Term) (c:Closure)
         (v:Res) (l:Lab) (xl:Varlist)
         (eval s a s' v) -> (In l (proj_lab_res v)) ->
         ((x:Var) (dummy x) /\ (fresh x xl) ->
                  (wrap (m x) (c x))) ->
         (t = (update (loc_in_res v l s') c s')) ->
         (eval s (over a l m) t v)
```

The *clone* semantics does not require any specific digression: again pointers are managed coherently with respect to the current store size. We use also the function *new_res_res* for generating a new result from an old one, similarly to the case of the object rule:

```
e_clone: (s,s',t:Store) (a:Term) (v,w:Res)
         (eval s a s' v) ->
         (t = (alloc s' (store_to_list (proj_loc_res v) s'))) ->
         (w = (new_res_res v (size s'))) ->
         (eval s (clone a) t w)
```

As explained in the previous chapter, the *method selection* requires the introduction of the extra predicate $eval_{body}$, used for evaluating closures, which applies to method bodies loaded in the store. The predicate $eval_{body}$ is defined by mutual induction together with the main *eval* predicate, and has two constructors. The first one corresponds to the case of method bodies without abstractions; the inductive constructor uses an hypothetical premise for evaluating a simpler closure in a context enriched by the binding of a fresh variable to the first result in the closure:

```
e_ground: (s,t:Store) (a:Term) (v:Res)
          (eval s a t v) ->
          (eval_body s (ground a) t v)

e_bind  : (s,t:Store) (c:Var->Body) (v,w:Res)
          ((y:Var) (stack y) = (v) ->
                   (eval_body s (c y) t w)) ->
          (eval_body s (bind v c) t w).
```

The method selection constructor uses the function *loc_in_res* for searching the right method name in the host object, then fetches the right closure in the store. Finally, it evaluates the closure body through the predicate $eval_{body}$, in a context enriched by a binding between a fresh variable and the (implementation of the) host object:

```
e_call : (s,s',t:Store) (a:Term) (v,w:Res) (c:Closure) (l:Lab)
         (eval s a s' v) -> (In l (proj_lab_res v)) ->
         (store_nth (loc_in_res v l s') s') = (c) ->
         ((x:Var) (stack x) = (v) ->
                  (eval_body s' (c x) t w)) ->
         (eval s (call a l) t w)
```

Static semantics.

We encode now the term typing system for **imp**ς. One of the crucial aspects of the NDS approach is the use of hypothetical premises, which implement fragments of the environment. As done for the stack in the operational semantics, we model the *typing environment* by introducing a function which has no implementation and is used for associating *object types* to variables:

```
Parameter typenv : Var -> TType.
```

Thus we work with the function `typenv` as extra knowledge in our derivation context.

Auxiliary functions. We have to implement some functions for manipulating the structure of object types. We need functions for projecting lists of labels from lists or object types ($proj\_lab\_list, labels$), for assembling and disassembling types ($insert, list\_from\_type$) and searching for labels ($type\_from\_lab$):

```
Fixpoint proj_lab_list [pl:(list (Lab*TType))] : (list Lab) :=
         Cases pl of nil => (nil Lab)
                  |  (cons (l,A) ql) => (cons l (proj_lab_list ql))
```

```
         end.
Definition labels : TType -> (list Lab) := [A:TType]
           Cases A of (mktype pl) => (proj_lab_list pl)
           end.

Definition insert : (Lab*TType) -> TType -> TType :=
           [a:(Lab*TType)] [A:TType]
           Cases A of (mktype pl) => (mktype (cons a pl))
           end.
Definition list_from_type : TType -> (list (Lab*TType)) := [A:TType]
           Cases A of (mktype pl) => (pl)
           end.
Fixpoint type_from_lab_list [pl:(list (Lab*TType))] : Lab -> TType :=
         [l:Lab]
         Cases pl of nil => (mktype (nil (Lab*TType)))
                  |  (cons (k,A) ql) =>
         Cases (eq_lab k l) of true => A
                            |  false => (type_from_lab_list ql l)
         end end.
Definition type_from_lab : TType -> Lab -> TType := [A:TType] [l:Lab]
           Cases A of (mktype pl) => (type_from_lab_list pl l)
           end.
```

Auxiliary judgments. We have shown in section 6.2 that we use in the NDS setting the two judgments:

$$\begin{array}{lcl} wt & \subseteq & TType \\ sub & \subseteq & TType \times TType \end{array}$$

The *well-formedness of types* has to be encoded taking care that object types are formed via lists; thus it is necessary to capture the *well-formedness of lists* as well:

$$wl \subseteq (list\ (Lab * TType))$$

The two judgments wt and wl are formalized by mutual induction:

$$\text{(wt\_mk)}\ \frac{(wl\ pl)}{(wt\ (mktype\ pl))}$$

$$\text{(wl\_nil)}\ \frac{}{(wl\ [\,])} \qquad\qquad \text{(wl\_cons)}\ \frac{(wl\ pl) \quad (wt\ A) \quad (l \notin \pi_1(pl))}{(wl\ ((l, A) :: pl))}$$

The check about well-formedness of types is delegated to the well-formedness of lists judgment. Empty lists are well-formed; longer lists are well-formed provided their tails are well-formed and the head component is built by a label different from all the other ones and a well-formed type. These rules are trivially formalized in Coq.

The *subtyping* relation on object types tells that longer types are subtypes of shorter ones, provided there is no variation in the common components:

$$\text{(sub\_obj)}\ \frac{(wt\ B_i) \quad (l_i\ distinct) \quad \forall i \in 1..n+m}{(sub\ [l_i : {B_i}^{i \in 1..n+m}]\ [l_i : {B_i}^{i \in 1..n}])}$$

The statement of the rule "on the paper" hides the possibility of *permutating* the component pairs and does not address the *invariance* of types associated to identical labels. Thus, in order to formalize the rule, it is necessary to characterize in a completely detailed way the notion of invariance for object types: that is, object types are neither covariant neither contravariant in their component types. Informally speaking, invariance is permutation plus invariance of types associated to identical labels. We have thus that object types are invariant if and only if each of them is subtype of the other one.

The characterization of subtyping suggested by the rule (*sub_obj*) is that, rearranging the pairs $(l_i, B_i)_{i \in I}$, there exists a prefix of the longer type coinciding with the shorter one, up-to invariance of the right-hand components of pairs, i.e. types. We see two possibilities for formalizing this:

- the subtyping verification is deterministic. First we permute the shorter type in such a way that the order of the labels coincides with the common ones with the longer type. After we scan and consume the two types checking that types associated to equal labels are invariant. We finish when the shorter type is completely scanned;
- the subtyping verification is not algorithmic. We scan and consume longer and shorter types in parallel but, at every moment, we can perform a permutation of the components of the shorter type. We conclude the process as above.

The first solution causes to introduce two predicates; the second one permits to work with only one notion, but is not deterministic. Anyway we prefer the second solution, because it is easier to use in a logical framework. So, using the auxiliary predicates *perm* (permutation) and *inv* (invariance), the (*sub_obj*) rule is equivalent to the following system of rules:

$$\text{(sub\_top)}\ \frac{(wt\ A)}{(A <: [\,]_T)} \qquad \text{(sub\_perm)}\ \frac{(A <: B) \qquad (perm\ B\ C)}{(A <: C)}$$

$$\text{(sub\_step)}\ \frac{(wt\ A) \quad (wt\ B) \quad (T <: U) \quad (inv\ A\ B) \quad (l \notin T) \quad (l \notin U)}{((l, A), T) <: ((l, B), U)}$$

We can capture permutations through the predicate *add*, i.e. random insertion:

$$\text{(perm\_refl)}\ \frac{}{(perm\ A\ A)}$$

$$\text{(perm\_add)}\ \frac{(perm\ A\ B) \quad (inv\ C\ D) \quad (add\ A\ (l, C)\ T) \quad (add\ B\ (l, D)\ U) \quad l \notin A, B}{(perm\ T\ U)}$$

$$\text{(add\_fst)}\ \frac{(inv\ A\ B) \quad l \notin T}{(add\ T\ (l, A)\ ((l, B), T))}$$

$$\text{(add\_dpt)}\ \frac{(add\ T\ (l, A)\ U) \quad (l \neq m) \quad m \notin T, U}{(add\ ((m, B), T)\ (l, A)\ ((m, B), U)}$$

As far as the invariance is concerned, it has to be modeled by an inductive equivalence relation on types: two types are invariant if and only if they are formed by the same

number of components with the same set of labels and, in turn, types corresponding to identical labels are invariant.

We remark that the formalization of subtyping is not central in the economy of our investigation, because the properties of the type system which are used in the proof of the Subject Reduction are very simple: the reflexivity and transitivity of subtyping and the properties stated in lemma 6.4 actually suffice. Hence the treatment of subtyping is far from the more involved arguments leading to the Subject Reduction. Thus we can formalize the invariance by the Leibniz's equality "=", namely an equivalence relation on types. By this choice, the invariance is flattened to the first level of object types, which is the same that Abadi-Cardelli do working with paper and pencil.

Term typing. The main aspect concerning the formalization of the term typing judgment is the treatment of objects. Since objects and object types are inductive structures, we have to type the methods one at time. By the rule (t_obj), in order to type all the methods of an object, it is necessary to carry along the whole type of the object itself. So we need an extra judgment, which is defined by mutual induction with the main term typing:

$$\begin{array}{lcl} type & \subseteq & Term \times TType \\ type_{obj} & \subseteq & TType \times Term \times TType \end{array}$$

We formalize and discuss the constructors of the two predicates. The constructors of the predicate *type* do not require specific explanations, since they are similar, but simpler, than the corresponding ones of the operational semantics:

```
Mutual Inductive type : Term -> TType -> Prop :=
    |     t_sub  : (a:Term) (A,B:TType)
                   (type a A) -> (subtype A B) ->
                   (type a B)
    |     t_var  : (x:Var) (A:TType)
                   (wftype A) -> ((typenv x) = A) -> (type x A)
    |     t_obj  : (ml:Object) (A:TType)
                   (type_obj A (obj ml) A) ->
                   (type (obj ml) A)
    |     t_call : (a:Term) (l:Lab) (A,B:TType)
                   (type a A) ->
                   (In l (labels A)) -> (type_from_lab A l) = (B) ->
                   (type (call a l) B)
    |     t_over : (a:Term) (m:Var->Term) (l:Lab) (A:TType)
                   (type a A) -> (In l (labels A)) ->
                   ((x:Var) (typenv x) = (A) ->
                            (type (m x) (type_from_lab A l))) ->
                   (type (over a l m) A)
    |     t_clone: (a:Term) (A:TType)
                   (type a A) -> (type (clone a) A)
    |     t_let  : (a:Term) (b:Var->Term) (A,B:TType)
                   (type a A) -> (wftype B) ->
                   ((x:Var) (typenv x) = (A) ->
                            (type (b x) B)) ->
```

```
                    (type (lEt a b) B)
```

The constructors of the predicate $type_{obj}$ reflect the inductive nature of objects:

```
with type_obj : TType -> Term -> TType -> Prop :=
       t_nil  : (A:TType)
                (type_obj A (obj (obj_nil))
                            (mktype (nil (Lab*TType))))
   |   t_cons : (A,B,C:TType) (ml:Object) (l:Lab)
                (m:Var->Term) (pl:(list (Lab*TType)))
                (type_obj C (obj ml) A) ->
                (wftype C) -> (type_from_lab C l)=B ->
                ((x:Var) (typenv x) = (C) ->
                         (type (m x) B)) ->
                (list_from_type A) = (pl) ->
                (wftype B) -> ~(In l (labels A)) ->
                (type_obj C (obj (obj_cons l m ml))
                            (mktype (cons (l,B) pl)))
```

The base constructor types the empty object with the empty object type; the induction constructor types (sub)objects starting from the typing of smaller subobjects. It is worth noticing that this solution of typing sub-objects can be useful in the perspective of encoding and typing extendible objects [AC96, GHL98, CDGHL02].

Result typing.

The formalization of the result typing does not require any new explanation with respect to the dynamic and static semantics. The essential aspect is that results are inductive structures, so we follow the approach used for typing objects in the static semantics, i.e. we carry along the whole (result) type while we scan and type the components of results.

Coinductive encoding. Since in Coq it is not possible to combine coinductive and inductive predicates in mutual definitions, we have to formalize *cores* and $cotype_{body}$ by mutual coinduction:

```
CoInductive cotype_body : Store -> Body -> TType -> Prop :=
         t_coground: (s:Store) (b:Term) (A:TType)
                     (type b A) ->
                     (cotype_body s (ground b) A)
   |     t_cobind  : (s:Store) (b:Var->Body) (A,B:TType) (v:Res)
                     (cores A s v A) ->
                     (wftype A) ->
                     ((x:Var) (typenv x) = (A) ->
                              (cotype_body s (b x) B)) ->
                     (cotype_body s (bind v b) B)

with         cores : TType -> Store -> Res -> TType -> Prop :=
          t_covoid : (A:TType) (s:Store)
                     (cores A s (nil (Lab*Loc))
```

```
                              (mktype (nil (Lab*TType))))
  |      t_costep : (A,B,C:TType; s:Store; v:Res; i:Loc; c:Closure)
                   (l:Lab) (pl:(list (Lab*TType)))
                   (cores C s v A) ->
                   (store_nth i s) = (c) -> (lt i (size s)) ->
                   (wftype C) ->
                   ((x:Var) (typenv x) = (C) ->
                            (cotype_body s (c x) B)) ->
                   (list_from_type A) = (pl) ->
                   (wftype B)-> ~(In l (labels A)) ->
                   ~(In i (proj_loc_res v)) ->
                   (cores C s (cons (l,i) v)
                              (mktype (cons (l,B) pl))).
```

Inductive encoding. When we move to the inductive result typing, we have to formalize store types. We choose the simplest possible solution, using pairs of object types:

```
Definition SType : Set := (list (TType * TType)).
```

We can easily define projection functions on store types and construct store types from object types:

```
Definition stype_nth : Loc -> SType -> (TType * TType) :=
           [n:Loc; S:SType]
           (nth n S ((mktype (nil (Lab*TType))),
                     (mktype (nil (Lab*TType))))).

Definition stype_nth_1 : Loc -> SType -> TType := [n:Loc; S:SType]
           Cases (stype_nth n S) of (A,B) => A
           end.

Definition stype_nth_2 : Loc -> SType -> TType := [n:Loc; S:SType]
           Cases (stype_nth n S) of (A,B) => B
           end.

Fixpoint scan_type [pl:(list (Lab*TType))] : TType -> SType :=
         [A:TType]
         Cases pl of nil => (nil (TType * TType))
                   |  (cons (l,B) ql) => (cons (A,B) (scan_type ql A))
         end.

Definition build_stype : TType -> SType := [A:TType]
           Cases A of (mktype pl) => (scan_type pl A) end.
```

Finally, the main judgments *res*, $type_{body}$, and *comp* are formalized as follows:

```
Inductive res : SType -> TType -> Res -> TType -> Prop :=
     t_void   : (S:SType) (A:TType)
                (res S A (nil (Lab*Loc)) (mktype (nil (Lab*TType))))
```

```
  |  t_step   : (S:SType) (A,B,C:TType) (v:Res) (i:Loc)
               (l:Lab) (pl:(list (Lab*TType)))
               (res S A v B) ->
               (stype_nth_1 i S) = (A) -> (stype_nth_2 i S) = (C) ->
               (lt i (dim S)) ->
               (wftype A) -> (type_from_lab A l)=C ->
               (list_from_type B) = (pl) ->
               (wftype C)-> ~(In l (labels B)) ->
               ~(In i (proj_loc_res v)) ->
               (res S A (cons (l,i) v) (mktype (cons (l,C) pl))).

Inductive type_body : SType -> Body -> TType -> Prop :=
          t_ground  : (S:SType) (b:Term) (A:TType)
                      (type b A) ->
                      (type_body S (ground b) A)
    |     t_bind    : (S:SType) (b:Var->Body) (A,B:TType) (v:Res)
                      (res S A v A) ->
                      ((x:Var) (typenv x) = (A) ->
                               (type_body S (b x) B)) ->
                      (type_body S (bind v b) B).

Definition dim : SType -> nat := [S:SType] (length S).

Definition ext : SType -> SType -> Prop := [S',S:SType]
           (le (dim S) (dim S')) /\
           (i:nat) (lt i (dim S)) -> (stype_nth i S')=(stype_nth i S).

Definition comp : SType -> Store -> Prop := [S:SType; s:Store]
           (le (size s) (dim S) /\
           ((i:nat) (lt i (dim s)) ->
              ((x:Var) (typenv x)=(stype_nth_1 i S) ->
                 (type_body S (store_nth i s x) (stype_nth_2 i S)))).
```

7.3 Metatheory in Coq

One of the main aims of the formalization presented in the previous section is to allow for the formal development of important properties of **imp**ς. The upmost important, yet delicate to prove, is the Subject Reduction, which states that the operational semantics is consistent with the type system. This result permits immediately to address the Type Soundness theorem, which assures that well-typed terms cannot get stuck during evaluation, and produce results of the expected type.

A lot of preliminary work has to be done in order to approach the Subject Reduction result in Coq. The essential steps towards the ultimate goal are the following:

- to address all the aspects concerning concrete structures specified, i.e. implemented, in the proof assistant;

- to establish the metatheory about the various fragments of the semantics: operational semantics, term typing and result typing;
- to extend in a suitable way the Coq environment with the Theory of Contexts;
- to address specific properties of the various constructs of the calculus that are used directly in the main theorem.

Proof techniques. The major differences between the Coq proofs and the "paper and pencil" ones of previous chapter depend on the use of *inductive structures* for encoding finite collections of components, as objects, object types, results, stores and store types. Therefore, several of the proofs about the properties of these explicit datatypes are carried out by induction on their structure, whereas they can be treated in a direct way on the paper.

As far as the *operational semantics* is concerned, the main structure which has to be worked with is the *store*: thus we have to prove a suitable collection of properties that permit to manage it and turn out to be sufficient in the following proof development. These properties typically involve the functions we have designed for handling the store, as *size*, *alloc* and *update*. Similarly, it is necessary to manage the results. The development of a starting package of lemmata is a long task, compared with the abstract statement of the operational semantics, that use only a mathematical language for expressing the intended meaning of the notions and operations. Thus in this phase is very useful the support given by the library `PolyList`, which permits to gain and use an initial set of lemmata about lists.

As far as the *type system* is concerned, we need to manage the object types and we have to address the relationship between the various judgments, i.e. *type*, *sub* and *wt*. Typically, it is necessary to search for labels in object types, to describe consequences of subtyping and to examine interactions between the predicates. The properties staying at the top level correspond essentially to the lemma 6.4, which can be proved in a standard way, arguing the proofs by structural induction on derivations.

When considering *result typing*, we have to distinguish between the two different approaches that we have designed in sections 6.5 and 6.6, which correspond, respectively, to **imp**ς without method update and the full calculus.

The coinductive encoding of result typing is more compact, because it permits to do without store types. The formal proofs about it correspond essentially to the lemmas 6.5, 6.6, 6.7 and 6.8, which have a direct impact on the proof of the Subject Reduction. It turns out that these lemmas are relatively easy to prove, because we have not to deal with store types. In particular, we have taken full advantage of the possibility of making coinductive proofs for the `cores` predicate via the `Cofix` tactic. The interaction between this predicate and $cotype_{body}$ is very interesting from a proof-theoretical point of view. In fact, these judgments are defined in Coq by mutual coinduction, but the nature of $cotype_{body}$ is inductive, because it is defined on the structure of closures. However, proofs about *cores* assertions become potentially infinite through the contribute of $cotype_{body}$, which may cause the need to type again the same result, thus forcing to reason coinductively.

On the other hand, in the inductive encoding of result typing, closer to the original setting of **imp**ς, the development of the metatheory is more involved, due to the handling of additional structures, i.e. store types. We remember that the inductive encoding has

been carried out in a second phase of our research, namely for managing the "method update" operator. It is important noticing that we are able to reuse some of the proofs developed for the coinductive encoding with a minimal effort. These proofs are those not requiring an explicit inspection on the structure of store types; in this case we have simply to convert potentially coinductive proofs carried out on derivations into proofs by induction on the structure of results. This re-usability of proofs witnesses the important fact that the present approach is quite modular.

However, some properties dealing with linear structures (such as stores) get much more involved when we consider also store types. In this case we cannot reuse part, neither follow the pattern, of the proofs carried out in the coinductive encoding. Instead, we have to develop different techniques, often more involved than every other one in the coinductive approach. This depends essentially on the simultaneous managing of explicit linear structures, typically stores and store types. This confirms that dealing with these structures in judgments is very cumbersome, and therefore that the choice of delegating the stack and typing environment to the metalinguistic proof context reduces considerably the length and the complexity of proofs.

Theory of Contexts. Another crucial aspect of our formalization is the well-known fact that HOAS presents some drawbacks. The main one is that most LFs do not provide an adequate support for higher-order encodings. For example, these systems neither provide recursion/induction principles over higher-order terms (i.e., terms with "holes") nor allow to access the notions related to the mechanisms delegated to the metalanguage. An important family of properties which cannot be proved in plain $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$ are the so-called *renaming lemmata*, that is, invariance of validity under variable renaming, such as the following, regarding types of terms and closure-bodies of **imp**ς:

```
Lemma rename_term : (m:Var->Term) (A:TType) (x,y:Var)
                    (type (m x) A) -> (typenv x)=(typenv y) ->
                    (type (m y) A).
Lemma rename_body : (S:SType) (c:Var->Body) (A:TType) (x,y:Var)
                    (type_body S (c x) A) -> (typenv x)=(typenv y) ->
                    (type_body S (c y) A).
```

In other words, the expressive power of LFs is limited, when it comes to reason on formalizations in (weak) HOAS.

A simple and direct way for recovering the missing expressive power is by assuming a suitable (but consistent) set of *axioms*. This is the approach adopted in [HMS01b, Sca02], where the Theory of Contexts (ToC), a simple axiomatization capturing some basic and natural properties of *(variable) names* and *term contexts*, is proposed. These axioms allow for a smooth handling of schemata in HOAS, with a very low mathematical and logical overhead, hence they can be plugged in existing proof environments without requiring any redesign of the system. Their usefulness has been demonstrated in several case studies on untyped and simply typed λ-calculus, π-calculus, and Ambients [Mic, HMS01a, SM02]. Therefore, the use of the ToC is a natural choice also in the present setting; it should be noticed that this is the first application of the ToC to a calculus featuring closures, objects and imperative features.

The Theory of Contexts consists in four axioms (indeed, axiom schemata):

unsaturation: $\forall M.\ \exists x.\ x \notin FV(M)$. This axiom captures the intuition that, since terms are finite entities, they cannot contain all the variables at once. The same axiom can be stated w.r.t. lists of variables, instead of terms, since it is always possible to obtain from a term the list of its free-variables: $\forall L.\ \exists x.\ x \notin L$;

decidability of equality over variables: $\forall x, y.\ x = y \vee x \neq y$. In a classical framework, this axiom is just an instance of the Law of Excluded Middle. In the present case, it represents the minimal classical property we need in an (otherwise) intuitionistic setting;

β-expansion: $\forall M, x.\ \exists N(\cdot).\ x \notin FV(N[\cdot]) \wedge M = N(x)$. Essentially, β-expansion allows to generate a new context N with one more hole from another context M, by abstracting over a given variable x;

extensionality: $\forall M(\cdot), N(\cdot), x.\ x \notin \{FV(M(\cdot), N(\cdot))\} \wedge (M(x) = N(x)) \Rightarrow M(\cdot) = N(\cdot)$. Extensionality allows to conclude that two contexts are equal if they are equal when applied to a fresh variable x. Together with β-expansion, extensionality allows to derive properties by reasoning over the structure of higher-order terms.

The above axioms are very natural and useful for dealing with higher-order terms as methods, closures and local declarations.

An important remark is that, in order to be effectively used for reasoning on **impς**, the "unsaturation" axiom has to be slightly modified with respect to the original formulation in [HMS01b].

The first difference is related to the presence of types. Similarly to the case of other typed languages, we assume informally that there are infinite variables for every type. This is equivalent to say that each variable "generated" by the unsaturation axiom can be associated to a given type.

The other difference is peculiar to **impς**, and it has not been needed in any previous application of the ToC, due to the presence here of implementation-level entities, such as *closures*. In the formalization of **impς**, variables are introduced in the Coq derivation context for two reasons: either during reductions and typing assignments (associated to results and types), or in the construction of closures (used just as place-holders). In the first case the new variable is associated both to results and types by the `stack` and `typenv` maps. In the second case, the new variable is marked as `dummy`, because it does not carry any information about results, but is used just as a typed place-holder, needed for typing closure-bodies in the store.

Thus, we observe a kind of "regularity" of well-formed contexts: for each variable x, there is always the assumption `(typenv x)=A` and, either `(stack x)=v` for some `v`, or `(dummy x)`. The unsaturation axiom has to respect this regularity; this is reflected by assuming two variants of unsaturation, one for each case, as follows:

```
Axiom unsat_res : (S:SType) (v:Res) (A:TType)
                  (res S v A) -> (xl:Varlist)
                  (EX x | (fresh x xl) /\ (stack x)=v /\ (typenv x)=A).

Axiom unsat : (A:TType) (xl:Varlist)
              (EX x | (fresh x xl) /\ (dummy x) /\ (typenv x)=A).
```

In `unsat_res`, the premise `(res S v A)` ensures the consistency between results and types associated to the same variable: this can be interpreted as the implementation of the original stack typing judgment. These axioms can be validated in models similar to those of the original ToC [BHH+01]. Moreover, this notion of regularity is close to the "regular world assumption" introduced by Schürmann [Sch01].

In order to convey better the above explanation to the reader, a typical example of the use of axiom `unsat` is for proving that the type of closure-bodies is preserved by the closure construction:

```
Lemma wrap_type : (A,B:TType) (m:Var->Term) (c:Var->Body)
                  (xl:Varlist) (s:SType)
                  ((x:Var) (typenv x)=A -> (type (m x) B)) ->
                  ((x:Var) (dummy x) /\ (fresh x xl) ->
                           (wrap (m x) (c x))) ->
                  ...
                  ((x:Var) (typenv x)=A -> (type_body S (c x) B)).
```

provided the maps `stack` and `typenv` are consistent w.r.t. the store-type S. The proof of this property requires the use of `unsat` and decidability of equality over variables. On the other hand, the proof of the lemmata `rename_term` and `rename_body` above requires the application of β-expansion and extensionality.

7.4 Type Soundness

We have listed and proved in the previous chapter a collection of lemmas, which are immediate used in the main proof of the Subject Reduction. It turns out that the proof in Coq of those results require a rephrasing, sometimes deep, of their statement. Correspondent proofs make use of those techniques, or similar ones, we have focused on in the previous section. After this preliminary phase, which is actually the heart of the formal development, we are in the condition of addressing the Subject Reduction theorem. The statements of the theorems 6.7 and 6.10 are formalized as follows:

Theorem 7.1 ***(Subject Reduction, coinductive setting)***

```
Theorem SR : (s,t:Store) (a:Term) (v:Res)
             (eval s a t v) ->
             (A:TType) (type a A) ->
             ((x:Var) (w:Res) (C:TType)
                      (stack x) = (w) /\ (typenv x) = C ->
                      (cores s w C)) ->
             (EX B:TType |
             (cores t v B) /\ (subtype B A)).
```

Theorem 7.2 ***(Subject Reduction, inductive setting)***

```
Theorem SR : (s,t:Store) (a:Term) (v:Res)
             (eval s a t v) ->
             (A:TType) (type a A) ->
```

```
(S:SType) (comp S s) ->
((x:Var) (w:Res) (C:TType)
         (stack x) = (w) /\ (typenv x) = C ->
         (res S w C)) ->
(EX B:TType | (EX T:SType |
(res T v B) /\ (ext T S) /\ (comp T t) /\ (subtype B A))).
```

Despite that `cores` is coinductive and `res` is inductive, both theorems are proved by structural induction on the derivation of `(eval s a t v)`. The formal development is documented at the web page of the author[1].

Type Soundness. The Type Soundness result is typically an immediate consequence of the Subject Reduction, hence it can be addressed in a canonical way.

7.5 Related work

In recent years, type theory-based proof assistants (as Coq, Lego, Alf) and generic theorem provers (as HOL, Isabelle, PVS) have emerged as promising specification and verification tools for the formal study of programming languages, as proved by several works in the literature. All these efforts, carried out using different approaches and studying the different programming paradigms, converge to address the synthesis of *certified software* for real world application languages.

The major research efforts for the object-oriented paradigm concern class-based languages. The Coq system has been used recently for formalizing the JavaCard Virtual Machine and studying formally the JavaCard Platform [BDJ+, BCDdS02], and for checking the behavior of a byte-code verifier for the Java Virtual Machine language [Ber01]. Restricting to metatheory, Coq has been used for proving the Type Soundness of a simply typed ML-like language with references [BD01].

Other systems have been used extensively for studying the Java programming language and related dialects: the more representative works are those using Isabelle and PVS. Isabelle, a generic theorem prover that can be instantiated to different logics, for example higher-order logic (HOL), has been employed for formalizing and certifying an executable bytecode verifier for a significant subset of the Java Virtual Machine [KN02]. PVS, a specification/verification system based on higher-order logic, and Isabelle itself are used as target systems for translations of coalgebraic specifications [RTJ00]. This technique has been applied to real programs in JavaCard [vdBJP01] and C^{++} [Tew00]. In these case studies, proof methods tailored to the particular languages have been developed, e.g. based on Hoare logic [Hui01].

It is worth noticing that, in front of applications to class-based languages, relatively little or no formal work has been done on object-based ones. Our investigation about **imp**ς, to our knowledge, is the first development of the theory of a object-based language with side effects, in Coq, using NDS, HOAS and coinduction.

[1] `http://www.dimi.uniud.it/~ciaffagl`

7.6 Conclusion

We have presented a formal development of the theory of **imp**ς, an object-based calculus with types and side effects, in the proof assistant Coq. We have tried to take most advantage of the features of the underlying coinductive type theoretic logical framework, namely $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$; therefore, we have developed an original presentation of **imp**ς, in the setting of Natural Deduction Semantics (NDS) and weak Higher Order Abstract Syntax (HOAS). This reformulation is interesting *per se*, since it allows for a simpler and smoother treatment of complex properties, such as Subject Reduction. In fact, for a significant fragment of **imp**ς, we have been able to eliminate "store types", in favour of *coinductive* typing systems. The complete system has been encoded in Coq, and the fundamental property of Subject Reduction formally proved.

The Natural Deduction Semantics approach, possibly with coinduction, is particularly well-suited with respect to the proof practice of Coq, also in the very stressing case of a non-trivial calculus based on objects. In fact the absence of the explicit environmental structures in the judgments has a direct impact on the structure of the proofs, reducing in particular their complexity.

The formalization of **imp**ς in Coq is part of a larger project involving the study, definition and certified implementation of a typed class-based language (of the SmallTalk family) and its intermediate object-based language (of the Self family) [Tea02]. A significant aim of the research is the development of certified tools for object-based, Self-like languages, as interpreters, type checkers and compilers [LM01, CFCL$^+$03]. For example interpreters can run on their own virtual machine or compiled on (hopefully) certified virtual machines, such e.g. the JVM one.

Future work. As a first step, we plan to experiment further with the formalization we have carried out so far. We will consider other interesting (meta)properties of **imp**ς, beside the albeit fundamental Subject Reduction theorem. In particular, we can use the formalization for proving observational and behavioural equivalences of object programs.

Then, we plan to extend our formal development to other object-based calculi, for instance calculi featuring *object extensions* [FHM94], or *recursive types* [AC96]. The latter case could benefit again from coinductive types and predicates.

From a practical point of view, the formalization of **imp**ς can be used for the development of *certified* tools, such as interpreters, compilers and type checking algorithms. Rather than *extracting* these tools from proofs, we plan to *certify* a given tool with respect to the formal semantics of the object calculus and the target machine. The literature reports some experiments concerning these advanced goals: Bertot uses the Coq system for certifying a compiler for an imperative language [Ber98], Strecker proves the correctness of a compiler from Java source language to Java bytecode in the proof environment Isabelle [Str02]. However, none of these works adopts higher-order abstract syntax for dealing with binders; we feel that the use of NDS and HOAS should simplify these advanced tasks in the case of languages with binders.

Bibliography

[AC96] M. Abadi and L. Cardelli. *A theory of objects*. Springer-Verlag, 1996.

[Acz88] P. Aczel. *Non well-founded sets*. In CSLI Lecture Notes 14, 1988.

[Bar92] H. P. Barendregt. *Lambda Calculi with Types*. In Handbook of Logic in Computer Science, Oxford University Press, 1992.

[BC90] H. J. Boehm and R. Cartwright. *Exact real arithmetic: formulating real numbers as functions*. In Research topics in functional programming, 1990.

[BCDdS02] G. Barthe, P. Courtieu, G. Dufay, and S. Melo de Sousa. *Tool-assisted specification and verification of the javacard platform*. In Proc. of AMAST, LNCS 2422, 2002.

[BD01] C. Dubois. *Proving ML type soundness within Coq*. In Proc. of TPHOLs, LNCS 1869, 2000.

[BDJ+] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, and S. Melo de Sousa. *A formal executable semantics of the javacard platform*. In Proc. of ESOP, LNCS 2028, 2001.

[Bee85] M. J. Beeson. *Foundations of Constructive Mathematics*. Springer-Verlag, 1985.

[Ber98] Y. Bertot. *A certified compiler for an imperative language*. INRIA Research Report RR-3488, 1998.

[Ber01] Y. Bertot. *Formalizing a JVML verifier for initialization in a theorem prover*. In Proc. of CAV, LNCS 2102, 2001.

[BH90] R. Burstall and F. Honsell. *Operational semantics in a natural deduction setting*. In Logical Frameworks, 1990.

[BHH+01] A. Bucalo, M. Hofmann, F. Honsell, M. Miculan, and I. Scagnetto. *Consistency of the theory of contexts*. Submitted, September 2001.

[Bis67] E. Bishop. *Foundations of constructive analysis*. McGraw-Hill, New York, 1967.

[BR99] D. Bridges and S. Reeves. *Constructive mathematics in theory and programming practice*. In Philosophia Mathematica 7, 1999.

[Bri99] D. Bridges. *Constructive mathematics: a foundation for computable analysis.* In TCS 219, 1999.

[Bro07] L. E. J. Brouwer. *Over de Grondslagen der Wiskunde.* PhD thesis, University of Amsterdam, 1907.

[Bro24] L. E. J. Brouwer. *Beweis, dass jede volle Funktion gleichmässig stetig ist.* In Proc. Amsterdam 27, 1924.

[Car95] L. Cardelli. *Obliq: A Language with Distributed Scope.* In Computing Systems, 1995.

[CDG00] A. Ciaffaglione and P. Di Gianantonio. *A co-inductive approach to real numbers.* In Proc. of Types, LNCS 1956, 2000.

[CDG01] A. Ciaffaglione and P. Di Gianantonio. *A tour with constructive real numbers.* In Proc. of Types, LNCS 2277, 2001.

[CDGHL02] A. Ciaffaglione, P. Di Gianantonio, F. Honsell, and L. Liquori. *Foundations for dynamic object re-classification.* Technical Report 03, Dipartimento di Matematica e Informatica, Università di Udine, 2003.

[Ced97] J. Cederquist. *A pointfree approach to Constructive Analysis in Type Theory.* PhD thesis, Göteborg University, 1997.

[CFCL+03] Cristal, Foc-CNAM, Lemme, Mimosa, Miró, and Oasis. *Concert: Compilateurs certifiés, 2003.* ARC INRIA 2003-2004, `http://www-sop.inria.fr/lemme/concert/`.

[CH88] T. Coquand and G. Huet. *The Calculus of Constructions.* In Information and Control 76, 1988.

[CH92] J. Chirimar and D. J. Howe. *Implementing constructive real analysis: preliminar report.* In Proc. of Constructivity in Computer Science, LNCS 613, 1992.

[Chu40] A. Church. *A formulation of the simple theory of types.* In Journal of Symbolic Logic, 1940.

[Chu41] A. Church. *The Calculi of Lambda Conversion.* Princeton University Press, 1941.

[Chu33] A. Church. *A set of postulates for the foundation of logic.* In Annals of Mathematics 2, 1932/33.

[CLM03] A. Ciaffaglione, L. Liquori, and M. Miculan. *Imperative object-calculi in (co)inductive type theories.* In Proc. of LPAR, LNAI (to appear), 2003.

[Con76] J. H. Conway. *On Numbers and Games.* In L.M.S. Monographs 6, 1976.

[Con86] R. L. Constable. *Implementing mathematics with the Nuprl development system.* Prentice-Hall, 1986.

[Coq93] T. Coquand. *Infinite objects in Type Theory.* In Proc. of Types, LNCS 806, 1993.

[CP90] T. Coquand and C. Paulin. *Inductively defined types.* In Proc. of COLOG, LNCS 417, 1990.

[Cur34] H. B. Curry. *Functionality in combinatory logic.* In Proc. of National Academy of Sciences, 1934.

[dB70] N. G. de Bruijn. *The mathematical language Automath, its usage and some of its extensions.* In Lecture Notes in Matehmatics, 1970.

[dB80] N. G. de Bruijn. *A survey of the project AUTOMATH.* In To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism. Academic Press, 1980.

[Des86] J. Despeyroux. *Proof of translation in natural semantics.* In Proc. of the First Conference on Logic in Computer Science. The Association for Computing Machinery, 1986.

[DFH95] J. Despeyroux, A. Felty, and A. Hirschowitz. *Higher-order syntax in Coq.* In Proc. of TLCA, LNCS 905, 1995.

[EH02] A. Edalat and R. Heckmann. *Computing with real numbers: (i) LFT approach to real computation, (ii) Domain-theoretic model of computational geometry.* In Gilles Barthe, Peter Dybjer, Luis Pinto, and Joao Saraiva, editors, LNCS, 2002.

[Esc00] M. Escardo. *Exact numerical computation.* Technical report, 2000.

[FGT90] W. Farmer, J. Guttman, and J. Thayer. *IMPS: an interactive mathematical proof system.* LNCS 449, 1990.

[FHM94] K. Fisher, F. Honsell, and J. C. Mitchell. *A lambda calculus of objects and method specialization.* In Nordic Journal of Computing, 1994.

[Geu92] H. Geuvers. *Inductive and Co-inductive types with iteration and recursion.* In Proc. of Types, 1992.

[Geu93] H. Geuvers. *Logics and Type Systems.* PhD thesis, Katholieke Universiteit, Nijmegen, the Netherlands, 1993.

[GHL98] P. Di Gianantonio, F. Honsell, and L. Liquori. *A lambda calculus of objects with self-inflicted extension.* In Proc. of OOPSLA, 1998.

[Gim94] E. Giménez. *Codifying guarded definitions with recursion schemes.* In Proc. of Types, LNCS 996, 1994.

[Gim98] E. Giménez. *Structural recursive definitions in Type Theory.* In Proc. of ICALP, LNCS 1443, 1998.

[Gir72] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[GN01] H. Geuvers and M. Niqui. *Constructive reals in Coq: axioms and categoricity.* In Proc. of Types, 2001.

[GPWZ00] H. Geuvers, R. Pollack, F. Wiedijk, and J. Zwanenburg. *The Fundamental Theorem of Algebra project.* Computing Science Institute, Nijmegen (The Netherlands), http://www.cs.kun.nl/∼freek/fta/index.html, 2000.

[Har96] J. R. Harrison. *Theorem proving with real numbers.* PhD thesis, Universitiy of Cambridge, 1996.

[HHP93] R. Harper, F. Honsell, and G. Plotkin. *A framework for defining logics.* In Journal of the ACM 40, 1993.

[Hir97] D. Hirschkoff. *Bisimulation proofs for the π-calculus in the Calculus of Constructions.* In Proc. of TPHOLs, LNCS 1275, 1997.

[HMS01a] F. Honsell, M. Miculan, and I. Scagnetto. *Pi-calculus in (co)inductive type theory.* In TCS 253, 2001.

[HMS01b] F. Honsell, M. Miculan, and I. Scagnetto. *An axiomatic approach to metareasoning on systems in higher-order abstract syntax.* In Proc. of ICALP, LNCS 2076, 2001.

[How80] W. A. Howard. *The formulæ-as-types notion of construction.* In To H.B. Curry: Essays on Combinatory Logic. Lambda Calculus and Formalism. Academic Press, 1980.

[Hui01] M. Huisman. *Reasoning about Java programs in higer order logic with PVS and Isabelle.* PhD thesis, Katholieke Universiteit Nijmegen, 2001.

[INR03] INRIA. *The Coq Proof Assistant V7.4.* `http://coq.inria.fr/doc/main.html`, 2003.

[J+98] B. Jacobs et al. *Reasoning about classes in Java.* In Object-oriented programming, systems, languages and applications, ACM Press, 1998.

[Jac96] B. Jacobs. *Objects and classes, co-algebraically.* In Object orientation with parallelism and persistence, Kluwer Acad. Publ., 1996.

[JHHT98] B. Jacobs, U. Hensel, M. Huisman, and H. Tews. *Reasoning about classes in object-oriented languages: logical models and tools.* In Proc. of ESOP, 1998.

[JMMZ01] V. Lefèvre, N. Revol, J.-M. Muller, G. Hanrot and P. Zimmermann. *Some notes for a proposal for elementary function implementation in floating-point arithmetic.* In Workshop IEEE 754R and Arithmetic Standardization (during Arith 15), 2001.

[Joh82] P. T. Johnstone. *Cambrdige studies in advanced mathematics.* Cambridge University Press, 1982.

[Jon91] C. Jones. *Completing the rationals and metric spaces in Lego.* In Logical Frameworks, Cambridge Universitiy Press, 1991.

[JR97] B. Jacobs and J. Rutten. *A Tutorial on (Co)Algebras and (Co)Induction.* In EATCS Bulletin, 1997.

[Jut77] L.S. Jutting. *Checking Landau's Grundlagen in the Automath system.* PhD thesis, Eindhoven University of Technology, 1977.

[Kah87] G. Kahn. *Natural Semantics.* In Proc. of the Symposium on Theoretical Aspects of Computer Science, LNCS 247, 1987.

[KN02] G. Klein and T. Nipkow. *Verified bytecode verifiers.* Theoretical Computer Science 298, 2002.

[KR35] S. C. Kleene and J. B. Rosser. *The inconsistency of certain formal logics.* In Annals of Mathematics 36, 1935.

[Len98] M. Lenisa. *Themes in final semantics.* PhD thesis, University of Pisa, 1998.

[Len99] M. Lenisa. *From set-theoretic coinduction to coalgebraic coinduction: some results, some problems.* In ENTCS 19, 1999.

[LM01] L. Liquori and M. Miculan. *Formal semantics and certified compilers for the language FunTalk.* Proposition pour équipes associés, `http://www.loria.fr/ lliquori/PAPERS/miro-slp.ps.gz`, 2001.

[MAC01] A. Momigliano, S. Ambler, and R. Crole. *A comparison of formalizations of the meta-theory of a language with variable bindings in Isabelle.* Technical Report 2001/07, University of Leicester, 2001.

[Mag95] L. Magnusson. *The implementation of Alf.* PhD thesis, Chalmers University of Technology, Göteborg, 1995.

[May01] M. Mayero. *Formalisation et automatisation de preuves en analyses réelle et numérique.* PhD thesis, Université Paris VI, 2001.

[Mic] M. Miculan. *Developing (meta)theory of lambda-calculus in the theory of contexts.* In Proc. of MERLIN, 2001.

[Mic94] M. Miculan. *The expressive power of structural operational semantics with explicit assumptions.* In Proc. of Types, LNCS 806, 1994.

[Mic97] M. Miculan. *Encoding local theories of programs.* PhD thesis, Dipartimento di Informatica, Università di Pisa, 1997.

[Mil83] R. Milner. *Calculi for synchrony and asynchrony.* In TCS 25, 1983.

[ML82] P. Martin-Löf. *Constructive mathematics and computer programming.* In Proc. of 6th international congress for logic, methodology and pholosophy of science, North Holland, 1982.

[NPS90] B. Nordström, K. Petersson, and J. M. Smith. *TProgramming in martin-Löf's type theory: an introduction.* International Series of Monograph in Computer Science. Oxford University Press, 1990.

[NS95] S. Negri and D. Soravia. *The continuum as a formal space.* Technical report, Dipartimento di Matematica Pura e Applicata, Università di Padova, 1995.

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. *PVS, a prototype verification system.* In Proc. of CADE, LNCS 607, 1992.

[Pau94] L. Paulson. *Isabelle: a generic theorem prover.* LNCS 828, 1994.

[PE88] F. Pfenning and C. Elliott. *THigher-order abstract syntax.* In Proc. of ACM SIGPLAN Symposium on Language Design and Implementation, 1988.

[PEE97] P. J. Potts, A. Edalat, and M. H. Escardo. *Semantics of exact real arithmetic.* In IEEE Symposium on Logic in Computer Science, 1997.

[Plo81] G. Plotkin. *A structural approach to operational semantics.* DAIMI FN-19, Computer Science Department, Århus University, 1981.

[PM93] C. Paulin-Mohring. *Inductive definitions in the system Coq: rules and properties.* In Proc. of TLCA, 1993.

[Pol94] R. Pollack. *The theory of Lego, a proof checker for the Extended Calculus of Constructions.* PhD thesis, University of Edimburgh, 1994.

[Rei95] H. Reichel. *An approach to object semantics based on terminal coalgebras.* In Mathamtical structures in Computer Science, 1995.

[RTJ00] J. Rothe, H. Tews, and B. Jacobs. *The coalgebraic class specification language CCSL.* Technical report, Dresden-Nijmegen, 2000.

[Rud92] P. Rudnicki. *An overview of the Mizar project.* Anonymous FTP from menaik.cs.ualberta.ca, 1992.

[Rut98] J. Rutten. *Relators and metric bisimulations.* In ENTCS 11, 1998.

[San95] D. Sangiorgi. *"On the proof method for bisimulation".* In Proc. of MFCS, LNCS 969, 1995.

[Sca02] I. Scagnetto. *Reasoning about names in Higher-Order Abstract Syntax.* PhD thesis, Dipartimento di Matematica e Informatica, Università di Udine, Udine, Italy, March 2002.

[Sch01] C. Schürmann. *Recursion for higher-order encodings.* In Proc. of CSL, LNCS 2142, 2001.

[Sim98] A. Simpson. *Lazy functional algorithms for exact real functionals.* In Proc. of MFCS, LNCS 1450, 1998.

[SM02] I. Scagnetto and M. Miculan. *Ambient calculus and its logic in the calculus of inductive constructions.* In Proc. of LFM, ENTCS 70.2, 2002.

[Str02] M. Strecker. *Formal verification of a Java compiler in Isabelle.* In Proc. of CADE, LNCS 2392, 2002.

[Tar55] A. Tarski. *A lattice-theoretical fixpoint theorem and its applications.* In Pacific Journal of Mathematics 5, 1955.

[Tea02] The Miró Team. *Proposition de Projet INRIA Miró.* Version v1.3, INRIA Lorraine & Sophia-Antipolis, 2002.

[Tew00] H. Tews. *A case study in coalgebraic specification: memory management in the FIASCO microkernel.* Technical report, 2000.

[TvD88] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics.* North-Holland, 1988.

[vdBJP01] J. van den Berg, B. Jacobs, and E. Poll. *Formal specification and verification of javacard's application identifier class.* In Proc. of the JavaCard Workshop, LNCS 2041, 2001.

[Wei00] K. Weihrauch. *Computable Analysis, An Introduction.* Springer-Verlag, 2000.

[Wer94] B. Werner. *Une théorie des constructions inductives.* PhD thesis, Université Paris 7, 1994.