Développement modulaire de théories et gestion de l'espace de nom pour l'assistant de preuve Coq.

Elie Soubiran

Modular development of theories and name-space management for the Coq proof assistant

THÈSE

présentée et soutenue publiquement le 27 Septembre 2010

pour l'obtention du

Doctorat de l'École Polytechnique

(spécialité informatique)

par

Elie Soubiran

Composition du jury

Directeur de thèse :	Hugo Herbelin
Rapporteurs :	Xavier Leroy
	Claudio Sacerdoti Coen
	Carsten Schürmann
Examinateurs :	Dale Miller
	Jacek Chrząszcz

Table of Contents

Abstract

Proof assistants offer a formal framework for formalizing and mechanically checking mathematical knowledge. Moreover, due to the numerous applications that follow from formal methods, the scientific production being formalized and verified by such tools is constantly growing. In that context, the organization and the classification of this knowledge does not have to be neglected. Coq is a proof assistant well-suited for program certification and mathematical formalization, and for seven years now it has featured a module system that helps users in their development processes. Modules provide a way to represent theories and offer a namespace management that is crucial for large developments. In this dissertation, we advance the module system of Coq by putting the emphasis on the two latter aspects. We propose to unify both module implementation and module type into a single notion of structure, and to split our module system in two parts. We have, on one hand, a namespace system that is able to define extensible naming scopes and to deal with renaming, and on the other hand a structure system that describes how to combine and to form structures. We define a new merge operator that, given two structures, builds the resulting structure by unifying components of the former two. In that dual system, a module is the association of a sub-namespace and a pair of structures, it acts as concrete declared theory. Furthermore, we adopt an applicative semantic for higher-order functors that allows a precise propagation of information. We show that this module system is a conservative extension of the underlying base language of Coq and we present the on-going implementation.

Acknowledgment

Introduction

Program correctness is a crucial need in an increasingly computerized society. It is now well admitted that specifying the behavior and the properties of programs is the first fundamental step of program developments. Furthermore, with the help of a formalized mathematical language, one can show that the refinement steps going from the specifications to the software realization preserve the desired behavior of the program. In this proof process, the computer can be a good ally for finding or verifying the demonstration of a property. Mechanized Mathematics Systems (MMS) are software that can answer to such needs. They implement logics well suited for the specification and the demonstration of programs. In the late 60's and early 70's, Nicolaas Govert de Bruijn and Robin Milner devised *Automath* and *LCF*, respectively. These programs lay the foundation of a new kind of MMS, called proof assistants. Since then, proof assistants have been an intensive research area and a wide variety of them has been implemented. For instance, Mizar, NuPrl and Coq are successors of *Automath*, whereas HOL, PVS and Isabelle are successors of *LCF*.

Today, these tools are mature enough to ensure the correctness of medium sized programs or protocols, and to formalize non trivial mathematics. As shown by some recent works or ongoing projects, such as the CompCert and Concurrent C Minor compilers [37, 29], the formal proof of the four colors theorem [23], or the flyspeck project[1], proof assistants can scale up to the "proving-in-the-large" challenge[2]. However, it still requires a great amount of work, and the development efforts need to be shared between a group of researchers. In that context, proof assistants need to provide tools to facilitate organization and reuse of existing mathematical development. There are different approaches to manage theories in a proof assistant. We extract from them two mainstream trends that are the axiomatic method used for instance in PVS, and module systems used for instance in Coq.

For now seven years, the Coq proof assistant has provided a module system. This module system has been formally studied and implemented by Jacek Chrząszcz [31, 11]. It is closely related to Objective Caml's (Ocaml) module system designed by Xavier Leroy [34, 36]. It extends the Coq proof assistant with module, signature, and functor constructions. Modules can package objects of the proof assistant, and hence give a meta notion of theory. Signatures are interface for modules. Functors act as functions from modules to modules. This module system has helped

[1] flyspeck: formal proof of the Kepler conjecture, ongoing project by Thomas C. Hales *et al.*

[2] By analogy to the "programming-in-the-large" challenge stated by Frank DeRemer and Hans Kron in [16].

Coq users to develop large modular libraries. However, this system has been originally designed for programming languages, and hence there are many perspectives of evolution in order to fit in the context of a proof assistant.

In this PhD dissertation, we propose an overhaul of the module system of Coq. The new module system consists of two parts. On one hand, we introduce a namespace system that is able to define extensible naming scopes and to deal with renaming. The originality of the namespace system is that it allows to build an alternative name-space from the one defined by the compilation units (i.e. modules) of the development. On the other hand, we define a structure system that describes how to combine structures. Here by notion of structure we refer to the unification of both notions of module implementations and module types[3]. In that context, a module is defined by the association of a sub-namespace and a pair of structures. In order to combine structures, we define a new merge of structures operator that, given two structures, builds the resulting structure by unifying components of the two formers. Finally, we adopt the OCaml applicative semantic for functors [35] in order to have a more precise propagation of typing informations at instanciation time. We formalize our module system in four steps where each step defines a system extending the previous one. In the first system, we extend the base language with the notion of namespace. In the second system, we add first-order modules by extending the first sytem with our new notion of structures. In the third one, we add a new merge operator on structures. In the final one, we consider higher-order structures and functors. By giving a translation from each system to its predecessor, we prove that our module system is a conservative extension of the base language of Coq.

The outline of this dissertation is organized as follows, in Chapter 1 we present some existing proof assistants together with their respective solution to manage theories. In Chapter 2, we focus on the Coq proof assistant, its original module system and our proposition to evolve it. In Chapter 3, we give an incremental formalization of our module system, where each steps is proved to be a conservative extension of the previous one. We propose in Chapter 4 interresting technical extensions of our module system that are a part of the concrete implementation. Then in Chapter 5, we describe the implementation of the module system in the Coq proof assistant. Finally we give concluding remarks and further research perspectives.

[3] Also known in ML as structures and signatures.

Chapter 1

Proof assistants and theory management

A Mechanized Mathematics System (MMS) is a fammily of software that offers a framework to formalize mathematical proofs. We distinguish two kinds of software in this family. The first one comes from the wish to have a fully automatic process of proof search. This can be done thanks to logics that are "sufficiently weak" to be automated. These tools are called automated theorem provers. The second one groups the proof assistants. They rely on more expressive and powerful logics that prevents automation. In those systems, the user builds a proof that will be checked by the proof assistant. Due to their expressiveness, proof assistants become more and more popular for mathematical formalizations and program verifications. In that context, the amount of knowledge being formalized and mechanically checked is constantly growing. Hence, proof assistants need to provide tools to facilitate organization and reuse of existing mathematical development. In this chapter, we investigate tools integrated in proof assistants that address this issue.

1.1 Formal Proofs and Proof Assistants

Proof assistants are a part of the MMS family, they are interactive programs that help the user to develop formal mathematics. They implement expressive logics that allow powerful reasoning scheme. In such frameworks, the user is able to specify a problem, propose a proof for it, that will finally be checked by the proof assistant. In order to build proofs, users have access to a set of procedures, called tactics, that construct pieces of proofs. Tactics can either be general or specific to a domain. For instance, a general tactic can be either a tactic that tries to apply an already proven lemma or a tactic that performs proof search with the respect of a database of applicable lemmas. In the other hand, a specialized tactic can be a tactic that proves equalities in the Presburger arithmetic.
Depending on the paradigm on which the proof assistant is built, the proof produced by tactics is either immediately accepted as correct, or is reverified by a proof-checker independent from the

tactics. These two different approaches are closely related to the design of LCF and Automath, respectively. Let us do a small review of some existing proof assistants. We present on one hand three proof assistants (IMPS, Isabelle, and PVS) based on the "LCF approach", and two provers (Agda, Coq) based on the Curry-Howard isomorphism.

IMPS implements a many-sorted simply type theory called LUTINS [20]. This type theory contains partial functions and allows subtypes to be included within types. Proofs in IMPS are interactively constructed using a natural style of inference based on the sequent calculus. IMPS builds a data structure, called a deduction graph, which records all the actions and results of a proof attempt. The user is only allowed to modify a deduction through the application of primitive inferences that are akin to inference rules. Most primitive inferences formalize basic laws of predicate logic and higher-order functions.

PVS is based on classical, typed higher-order logic [44]. The system allows the definition of predicate subtypes that can be used to introduce constraints on types (e.g. the type of prime numbers). These constrained types may incur proof obligations (TCCs) during type-checking. As in IMPS, PVS provides a collection of primitive inference procedures that are applied interactively within a sequent calculus framework. User-defined procedures can combine these primitive inference procedures. Proofs constructed from these tactics are accepted without rechecking.

Isabelle is a generic prover, it implements a meta-logic called *Isabelle/Pure* that allows the formalization of the syntax and inference rules [57]. The most commonly used instantiation is *Isabelle/HOL*, for Higher-Order Logic. The core logic is implemented according to the so-called "LCF approach", hence proofs can only be constructed by a small set of primitive inference rules. However, explicit proof terms are also available and can be checked by independent proof checker [7]. The proof language of Isabelle is called Isar, it is a declarative language which aims to be a tradeoff between primitive proof objects and the natural language.

Agda implements an extension of Martin-Löf type theory [9]. The main concern of Agda is to provide a dependently typed functional programming language. Thanks to the Curry-Howard isomorphism and dependent types, Agda can be used as a proof assistant, allowing to prove mathematical theorems and to run such proofs as algorithms. Hence, writing a proof in Agda is done as writing a program.

Coq is also a proof assistant based on the Curry-Howard isomorphism [17]. It implements the predicative Calculus of Inductive Constructions. Hence, Coq supports polymorphism, dependent types, inductive type definitions and a hierachy of type universes. Note that the latter is hidden to the user through the implementation of the typical ambuiguity [30, 27]. Proofs are objects of the underlying logical formalism and they are built with the help of tactics. Once built, proofs are

checked by an independent type-checker, called the kernel. Note that there are other proof assistants that implement the Calculus of Inductive Constructions such as Matita [3, 2] and Lego [19].

Let us say a few extra words about the Curry-Howard isomorphism. In 1958 Curry noticed that typing derivations in simply typed lambda calculus corresponds to natural deduction proof in minimal propositional logic. Moreover, in 1965 Tait saw the correspondence between the computation rule of the lambda calculus, and cut elimination. More precisely, it turned out that normal proofs correspond to lambda terms in normal form and that the cut elimination theorem corresponds to normalization property of the simply typed lambda calculus. Quite a few more logics have their lambda calculus counterparts. For instance, second-order propositional logic corresponds to Girard's system F. This phenomenon is called the Curry-Howard isomorphism: it establishes the correspondence between propositions and types, proofs and terms, provability and inhabitation, proof normalization and term reduction. In a proof assistant based on the Curry-Howard isomorphism such as Coq or Agda, stating a theorem amounts to writing a well-formed type, and proving this theorem is nothing else but finding a term, whose type corresponds to the theorem in question.

1.2 Theory management in Proof Assistants

The organization and the classification of the knowledge is a human habit. Let us take, the illustration of a library to introduce the basic concepts that we need to manage mathematical development in a proof assistant. In a library, books are grouped by topic, sub-topic and books themselves are divided into several chapters. Suppose that we want to consult a book on group theory. We are in the library, we go to the mathematical department, we find the algebra bookcase, then the row of group theory, and finally we take a book called "the handbook of group theory". We can first remark here that we have followed a path where each step has a name. Hence our first essential concept is the faculty to name things and furthermore names should be able to represent a path from a general set of knowledges to a specific one. Now, with the help of our book, we want to check if the set of permutations of three letters together with the composition of permutation defines a group. Unless we are lucky, we will not find this specific group in our book. In fact we find an abstract definition of a group: a set equipped with a binary operation that must satisfy some axioms. In the book, all the theorems or lemmas rely on this abstract definition which can be refined or extended for some specific properties (e.g. the cardinal of the set must be a prime number). If we want to benefit from all the properties stated in the book for our concrete example, we need to instantiate the abstract definition of groups. Concretely, it means that we give the definition of the set, the definition of the binary operation and then we prove the axioms. This example gives us our second essential concept which is composed by both abstraction and instantiation and as a consequence theorem reuse.
Mostly all proof assistants take care of providing tools to manage developments. There exists different approaches to solve this issue. Here, we extract two mainstream trends. The first one is

the axiomatic method that comes from the mathematical world. The second one is the module system that comes from the programming language world. As we will see these approaches answer differently the challenge of theory management.

1.2.1 The Axiomatic Method

The aim of the axiomatic method is to represent a mathematical model as a set of axioms expressed in a given language. Take for instance, the Peano axioms in a second-order logic language. Hence, a theory is given by a formal language (a logic) and a set of axioms. Theorems of a theory are logically derived from its axioms. As far as we know, the organization and the presentation of the mathematics with the axiomatic method take its basis in the Euclid's *Element*. Indeed, he gives an axiomatic presentation of Euclidean geometry and number theory. Many centuries later (1800s), the axiomatic method regained interest. For instance, take the *Principia Mathematica* of A. Whitehead and B. Russell, or *The Foundation of Geometry* of Hilbert.

Nowadays, it is commonly admitted that they are two different ways of using the axiomatic method, and somehow the two previous instances highlight it. They are generically called the *little theory* and the *big theory* approaches of the axiomatic method [21]. In the big theory approach, a body of mathematics is entirely represented in one powerful and highly expressive theory, and all reasoning is performed within this theory. It is the approach followed in *Principia Mathematica*, indeed the authors attempted to derive all mathematical truths from a well-defined set of logical axioms. Today, a big theory appreciated by mathematicians is Zermelo-Fraenkel set theory. In the little theory approach, a body of mathematics is represented as a network of theories. Bigger theories are composed of smaller theories and theories are linked by interpretations. In this approach the reasoning is distributed over the network of theories. A theory interpretation is a syntactic translation between a source theory to a target theory. This translation maps axioms of the source theory to theorems of the target theory. Interpretations fulfill different purposes. First it allows theorems of the source theory to be reused in the target theory. Second, a theory interpretation from a theory $\mathcal{T}$ to a theory $\mathcal{T}'$ demonstrates the consistency of $\mathcal{T}$ relative to $\mathcal{T}'$, and also the decidability of $\mathcal{T}$ modulo the decidability of $\mathcal{T}'$ [56]. Finally, it can be used to establish if a formula is independent from a given theory. In the *The Foundation of Geometry*, Hilbert uses the little theory approach to study relations of independence among subsets of axioms.

Theory and theory interpretation in proof assistants

Imps implements the little theory approach [21]. An IMPS theory is constructed from a (possibly empty) set of subtheories, a language, and a set of axioms. The subtheory relation and theory interpretation are ways to relate theories. The subtheory relation is induced by theory extensions. The theory interpretation coincides with the definition given before, unless that the translation also generates a set of obligations that must be proven as theorems of the target

theory. The following examples illustrate some relations on theories definable in IMPS:

- *Extension:* the theory of monoids can be extended to the theory of groups.
- *Interpretation:* one builds a theory interpretation, from the metric space theory to the normed space theory, by interpreting the distance function of the metric space as the function $(x, y) \rightarrow ||x - y||$.
- *Interpretation as instantiation:* considering a theory of abstract module over a ring and a theory of real arithmetic (formalized as a field), one obtains a theory of abstract real vector by instantiating the ring of the module theory with the field of reals.

A PVS theory consists of a theory name, a list of formal parameters, an assumption part, and a theory body [45]. The theory body is the main part of the theory, consisting of top-level theory importations, axioms, and theorems. The assumption part gives constraints over the current theory, which have to hold for any instance of the theory. Internally, the assumptions are the same as axioms. Externally, they generate obligations which must be proved for each import of the theory.

The importation mechanism is similar to the extension mechanism of IMPS. PVS provides two notions related to theory interpretation: *parameterized theory* and *explicit theory interpretation*. A parameterized theory in PVS is parameterized by generic types and constants specified in the formal parameter list. A generic type parameter is an uninterpreted type. Axioms are simulated by the assumptions over these formal parameters. This provides a way of presenting abstract theories. For instance, an abstract theory of groups is defined as follows:

```
group[G: TYPE, ∘ :  [G, G -> G], e:  G, inv:  [G -> G]]: THEORY
 BEGIN
  ASSUMING
  a, b, c:  VAR G
  assoc :  ASSUMPTION a ∘ (b ∘ c) = (a ∘ b) ∘ c
  ...
  ENDASSUMING
  leftcancellation :  THEOREM . .  .
  ...
 END group
```

An instance of a group can be imported by providing all actual parameters of group theory corresponding to the formal parameters, like `group[int, +, 0, -]` for a concrete instantiation

over natural numbers. This actually supplies a translation from the abstract (source) theory to the concrete (target) theory. The assumptions are then checked to generate TCCs (type correctness conditions) which are essentially proof obligations. Once the TCCs are proved, an interpretation is built and all interpreted theorems of the abstract group theory are available in the (target) theory importing it.

The explicit interpretation mechanism is similar to the theory parameterization mechanism, where axioms replace the corresponding assumptions. An explicit mapping is specified for uninterpreted types and constants of the source theory into the current theory. The interpreted theorems are considered proved and available for use if they are proved in the abstract theory. The group example above can then be reformalized as follows:

```
group:  THEORY
 BEGIN
  G: TYPE
  ∘:  [G, G -> G]
  ...
  assoc :  AXIOM FORALL a, b, c:  a ∘ (b ∘ c) = (a ∘ b) ∘ c
  ...
 END group
```

and an explicit interpretation over the group of natural numbers is written now `group{{G := int, + := +, 0 := 0, - := -}}`. One advantage of using mappings instead of parameters is that not all uninterpreted entities need to be mapped at import time, whereas for parameters either all or none must be given. On contrary IMPS, PVS has also extended parametric theories to take theories as parameter. For example, we may have a theory `group_homomorphism` of group homomorphism that takes two groups as parameter:

```
group homomorphism[G1, G2:  THEORY group]:  THEORY
 BEGIN
  x, y:  VAR G1.G
  f:  VAR [G1.G -> G2.G]
  homomorphism?(f):  bool = FORALL x, y:  f(x ∘ y) = f(x) ∘ f(y)
  ...
 END group homomorphism
```

One notes that symbol names are qualified by the theory names (e.g. `G1.G` and `G2.G`). Effectively, two copies of the group theory are parameters of the generic theory group homomorphism. Theory interpretations are used as parameter passing mechanisms. Two different theory interpretations can be provided as instantiation of the theory group homomorphism.

Isabelle The theory concept for Isabelle is locales. They were initially designed to provide scopes that locally fix variables, assumptions, definitions and pretty-printing syntax [33]. For instance, we define a locale that fixes a context for partial order:

```
locale partial_order =
  fixes le ::  "'a ⇒ 'a ⇒ bool" (infixl "⊑" 50)
  assumes refl [intro, simp]:  "x ⊑ x"
    and anti_sym [intro]:  "[| x ⊑ y; y ⊑ x |] ⟹ x = y"
    and trans [trans]:  "[| x ⊑ y; y ⊑ z |] ⟹ x ⊑ z"
```

This locale has a parameter `le` and an implicit type parameter `'a` and it declares the axioms for reflexivity, antisymmetry and transitivity of `le`. A locale definition also generates a locale predicate, which is basically a predicate that represents the locale in the meta-logic of Isabelle. On the contrary of PVS or IMPS, definitions and theorems can be added *a posteriori* in the locale. For instance, we add to the locale the following definition:

```
definition (in partial_order)
  less ::  "'a ⇒ 'a ⇒ bool" (infixl "⊏" 50)
  where "(x ⊏ y) = (x ⊑ y ∩ x ≠ y)"
```

This definition adds at top-level the definition `partial_order.less` lifted over the locale predicate and it also adds the conclusion of the definition as `less_def` in the locale. As in PVS, locales have their own notion of locale extension (aka import). For instance, one extends the locale `partial_order` to a `lattice` or to a `total_order` locales.

```
locale lattice = partial_order +
   assumes ex_inf:  "∃ inf.  is_inf x y inf"
   and ex_sup:  "∃ sup.  is_sup x y sup"
```

```
begin
definition meet ...
definition join ...
end
```

This gives the possibility of defining a hierarchy of locales [4]. One notes that the extension of locale is generalized into a merge of locale expressions. Locale expressions are either a name of a locale (as in the example) or a name of the locale plus some renaming. if `l1` and `l2` are two locale expressions then `l1+l2` denotes the merging of the two locales. At this stage, the hierarchy of locale is basically a DAG where each edge represents the extension of a source locale to a target locale. Since the notion of locale is very similar to the notion of little theory, a theory interpretation for locale has been added to Isabelle [5]. This locale interpretation is achieved through two commands. The first, `sublocale` interprets a source locale in the context of a target locale. The other, `interpretation` instantiates parameters and replaces definitions of a locale. For instance, a total order can be seen as a lattice:

```
sublocale total_order ⊆ lattice
```

For this interpretation we need to prove that the assumptions of the lattice are provable in the context of a total order. However, the assumptions that comes from the partial order locale are discharged since both source and target locales inherit from the partial order locale.

The instantiation of a locale is achieved through the `interpretation` command. For example, we instantiate the partial order locale on the natural numbers, simultaneously we give an effective value for the `le` parameter and replace the `less` definition.

```
interpretation nat:  partial_order "op ≤ ::  [nat, nat] ⇒ bool"
  where "partial_order.less op ≤ (x::nat) y = (x ≤ y)"
```

Once again, proof obligations are generated to checkout if the actual instantiation fulfill the conditions imposed by the axioms. As a matter of facts, interpretation links between locales entail cyclicity in the hierarchy of locales. In that context, the dependency graph of locales is maintained with the help of a development graph [4]. It helps to discharge automatically proof

obligations implied by the dependencies (cf. `sublocale` example) and it manages dynamically the propagation of theorems through dependencies. Finally, the addition of constructive type classes, that works well in combination with with locales, increases the modularity of the system.

1.2.2 Module Systems

Module systems are more recent and come from the world of programming languages. Their very first role in programming language was to answer the challenge of programming-in-the-large. The approach followed by module systems is closely related to the old adage "Divide et impera". Indeed, modules allow to split a large program development into several pieces of code as independent and as general as possible. In that sense, a module system achieves the separation of concerns for program developments. A module is typically a pair formed by an interface and an implementation. An interface expresses the elements that are provided and required by the module. The implementation gives the realization of the elements declared in the interface. Classical examples of imperative programming languages that provide such module systems are: C, the Modula family or Ada. Although, Ada [55] and Modula-3 [10] offer supports for generic modules (e.g. modules abstracted over types or values), they can only perform some basic operations on modules. On the other hand, most all ML dialects [42, 36] (Ocaml, SML/NJ, Moscow ML...) offer a much more powerful module language. The basic constructions of ML module systems are structures, signatures and functors. A structure is a sequence of type, value, and substructure bindings. A signature serves as interface for a structure, it is a list of specifications that assigns a type for each value component and either an abstract or manifest type specification for each type component. For instance take the following structure and signature in SML syntax:

```
signature Int_Set =
  sig
    type elem = int
    type set
    val emptyset :  set
    val insert :  elem -> set -> set
    val mem :  elem -> set -> bool
    ...
  end

structure IntSet :  Int_Set =
  struct
    type elem = int
    type set = int list
    let emptyset = []
```

```
  let insert e s = e::s
  let mem e s = ...
  ...
end
```

The signature `Int_Set` contains a manifest type specification `type elem = int` and an abstract one `type set`. The structure `IntSet` is sealed by the signature `Int_Set` (i.e. `: Int_Set`), and provides an implementation for all mentioned component in it. In order to use such a module, one has to project out the component using the dot notation (e.g. `IntSet.t`). Since, the type `set` is specified abstractly in the signature, clients of the module will only be able to build values of type `set` by using the associated values and functions (i.e. `emptyset`, `insert`...). This kind of signatures mixing abstract and manifest type specifications are called translucent signatures [24, 34]. Together with opaque sealing, they enable a good trade-off between data abstraction and type information propagation.

Functors are functions from modules to modules. They allow to build modules that depends on module parameters, and hence they enable modules to be reused on different instantiations of its parameters. If we go back to the previous `IntSet` example, we note that the implementation of set is quite independent from the type of the elements stored in the sets. Hence, we can give a generic version of the implementation of sets over the type of the elements if and only if the elements are comparable. First, we define a signature that specifies the minimal information needed to build an implementation of sets:

```
signature Ordered_Type =
  sig
    type t
    val cmp :  t -> t -> order
  end
```

Then, we define the functor `Set` as the structure depending on a parameter of signature `Ordered_Type`:

```
functor Set (X : Ordered_Type) =
  struct
    type elem = X.t
```

```
  type set = elem list
  let emptyset = []
  let insert e s = e::s
  let mem e s = ...  if X.cmp...
  ...
end
```

Now having a generic implementation of sets, the client can instantiate the functor on a concrete module that matches the `Ordered_Type` signature. One can refer to Appendix A for a more detailed presentation and history of the ML module system.

Now, we do a review of proof assistants that use a module system to manage theory:

Agda The module system of Agda is very simple. It has for main purpose the handling of name-space. In a sense it looks like the module systems of Modula-2 or Ada. A module is defined by a list of parameters and a set of definitions and submodules. For instance:

```
module Main where

    module N where
      data type Nat :  Set where
        Zero :  Nat
        Suc :  Nat -> Nat
      IsZero :  Nat -> Set
        IsZero Zero = True
        IsZero (Suc n) = False

    module L (A : Set) where
      datatype List :  Set where
        nil :  List
        cons :  A -> List -> List
```

The indentation delimits the scope of a module. In order to access to a field of a module, one has to use the dot notation (e.g. `N.IsZero`). The scope of a module can be opened, and thus making accessible its content without qualifying it. Parameterizing a module has the effect of abstracting the parameters over the definitions in the module. However, parameters can

also be globally instantiated. For instance, in the scope of the module main, one can write: `module M = L N.Nat`. Finally, Agda proposes two name modifiers, one for hiding the name of a field, and the other for renaming it. For instance, while opening a module one can rename fields as following: `open N renaming (IsZero to foo)`.

Lego The module system of Lego [48, 47] developed by Randy Pollack is roughly the same as the one defined in Agda. The most relevant difference is that modules are considered as first-class objects. However, they have a unique uninformative type, namely THEORY. Hence the only thing that can do a function that takes a parameter of type THEORY is to give it back. As in Agda, the module system of Lego is essentially used for name-space management.

Coq The Coq proof assistant offers different vectors of modularity for mathematical developments that are: a module system [12], a type class mechanism à la Haskell [54], and dependent records in conjunction with coercion graphs [51, 22]. One notes that the two last features are derived from inductive types constructions and hence are first class objects. The module system is basically the Ocaml module system with a generative functor semantic (we give a more precise presentation in Chapter 2). As expected, it allows to define signatures, modules and functors. These constructions are able to package heterogeneous objects. In other words, they can contain regular definitions, axioms, inductive type definitions, but also non-logical objects (e.g. pretty-printing syntax, databases of hints for proof search...). Coq has its own compiler that produces bytecode. Hence, another interesting contribution of the module system is that modules are considered as compilation unit and thus it provides mechanisms to quickly load libraries. The main drawback of this module system is that modules are not first-class values. Indeed, the system is built on top of the base language of Coq, and consequently the module language is clearly separated from the base language. However, we can put in perspective this limitation. The other vectors of modularity mentioned above can substitute modules when first class objects are needed. Furthermore, the type-checking algorithm of the base language is certified and need to stay relatively simple. In view of this fact, extending module computation to make module first class as done in [24][4], or adding packaged module constructions into terms of the base language as done in [50] would complicate the certification and would decrease the reliability of Coq.

1.3 Conclusion

We have presented different proof assistants together with their respective solutions to manage theories. These solutions can be classified in two main families that are the axiomatic method and the module system. When theorems are dynamically propagated through interpretation links, as in Isabelle, the axiomatic method seems to be a better tool for theorem reuse than module systems. On the other hand, in PVS or IMPS, interpretations are equivalent to first-order functors. Module systems are more adequate for the management of the name-space and the non-logical

[4] moreover this approach can lead to undecidability of type-checking.

features of the proof assistant. Indeed, the clear separation between module constructions and base language constructions allows to have a meta notion of theory. Consequently, it enhances the proof assistant with features that are orthogonal from the base language. Finally, modules are also useful for the development of libraries since they can be used as compilation units.

Chapter 2

The Coq proof assistant

In this chapter, we focus on the Coq proof assistant, its underlying logic language and its module system. Afterwards, we discuss about limitations of the actual module system and give our solutions to make it more "proof assistant compliant".

2.1 The formal language of Coq

2.1.1 Pure Type Systems

We give the definition of Pure Type System (PTS) introduced by Terlouw and Berardi [6]. Given $\mathcal{C}$ a countable set of constants, and $\mathcal{V}$ a countable set of variables, one defines the syntactic class of pseudoterms for pure type systems as follows:

$$t, T := v \mid (t_1\, t_2) \mid \lambda v : T.t \mid \forall v : T_1.T_2 \mid c$$

Concretely, a pseudoterm is either a variable, an application of two pseudoterms, an abstraction, a product, or a constant. One defines the notion of environments, written E, as a finite sequence of declarations of the form $v : T$.
Then, one defines the notion of β-reduction, written $\rhd_\beta$, as the following relation:

$$(\lambda v : T, t)\, t_1 \rhd_\beta \{v/t_1\}t$$

and one denotes $=_\beta$ the transitive, symmetric, reflexive and congruent closure of this relation. Finally, one defines the triple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ where:

- $\mathcal{S}$, the set of sorts, is a subset of $\mathcal{C}$.
- $\mathcal{A}$, the set of axioms, is composed of pair $(s, s') \in \mathcal{S} \times \mathcal{S}$.
- $\mathcal{R}$, the the set of rules, is a set of sort triplet $(s_1, s_2, s_3) \in \mathcal{S} \times \mathcal{S} \times \mathcal{S}$.

A PTS is characterized by the triple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$, and by the following typing system that defines the two judgements $E \vdash \text{ok}$ and $E \vdash t : T$.

$$\textsc{Env/Empty}\ \frac{}{\vdash \text{ok}} \qquad \textsc{Env/Acc}\ \frac{E \vdash \text{ok} \qquad (v : T) \in E}{E \vdash v : T} \qquad \textsc{Env/Insert}\ \frac{E \vdash \text{ok} \qquad E \vdash T : s \qquad s \in S}{E; (v : T) \vdash \text{ok}}$$

$$\textsc{Term/Ax}\ \frac{E \vdash \text{ok} \qquad (s_1, s_2) \in \mathcal{A}}{E \vdash s_1 : s_2} \qquad \textsc{Term/App}\ \frac{E \vdash t : \forall v : U, T \qquad E \vdash u : U}{E \vdash (t\, u) : \{v/u\}T}$$

$$\textsc{Term/Lambda}\ \frac{E \vdash \forall v : T, U : s \qquad E; (v : T) \vdash t : U}{E \vdash \lambda v : T, t : \forall v : T, U} \qquad \textsc{Term/Prod}\ \frac{E \vdash T : s_1 \qquad (s_1, s_2, s_3) \in \mathcal{R} \qquad E; (v : T) \vdash U : s_2}{E \vdash \forall v : T, U : s_3}$$

$$\textsc{Term/Conv}\ \frac{E \vdash U : s \qquad E \vdash t : T \qquad E \vdash T =_\beta U}{E \vdash t : U}$$

The judgement $E \vdash \text{ok}$ means that the environment E is well-formed (i.e. it assigns correct types to variables). The judgement $E \vdash t : T$ means that t is of type T in E.

Adding Definitions

Adding the possibility to abbreviate terms in PTS is a natural extension and has been formally studied in [52]. Concretely, one adds new entries of the form $(v : T := t)$ in the environment. One also defines the two following rules to add and access a definition in the environment:

$$\textsc{Env/Acc}\ \frac{E_1; (v : T := t); E_2 \vdash \text{ok}}{E \vdash v : T} \qquad \textsc{Env/Insert}\ \frac{E \vdash \text{ok} \qquad E \vdash t : T}{E; (v : T := t) \vdash \text{ok}}$$

In order to use the definitions, one defines the new reduction rules δ:

$$E_1; (v : T := t); E_2 \vdash v \rhd_\delta t$$

and one denotes $=_\beta$ the transitive, symmetric, reflexive and congruent closure of this relation. Consequently, the conversion rule becomes:

$$\textsc{Term/Conv}\ \frac{E \vdash U : s \qquad E \vdash t : T \qquad E \vdash T =_{\beta\delta} U}{E \vdash t : U}$$

Adding Inductive Types

An inductive type definition is a package containing names and types of mutually defined inductive types and constructors [46, 13]. One denotes an inductive definition as a pair of sequences of declarations: $(E^I := E^c)$, where $E^I = v_1 : T_1; \ldots; v_k : T_k$, and $E^c = k_1 : U_1; \ldots; k_n : U_n$. The rules that describe the introduction of an inductive type definition are the followings:

ENV/INSERT/IND

$$\frac{E \vdash T_j : s_j \qquad \forall j \in [1 \ldots k] \qquad E; E^I \vdash U_q : s_q \qquad \forall q \in [1 \ldots n]}{E; (E^I := E^c) \vdash \text{ok}} \text{ if } \text{POS}(E^I := E^c)$$

ENV/ACC/IND

$$\frac{E_1; (E^I := E^c); E_2 \vdash \text{ok}}{E_1; (E^I := E^c); E_2 \vdash v_i : T_i}$$

ENV/ACC/CONSTRUCTOR

$$\frac{E_1; (E^I := E^c); E_2 \vdash \text{ok}}{E_1; (E^I := E^c); E_2 \vdash k_i : U_i}$$

where the side condition $\text{POS}(E^I := E^c)$ is a "positivity" predicate syntactically defined on the kinds of inductive types and types of constructors. In this Phd dissertation, we restrict our description of inductive type definitions to constructions that are needed for modules. This is why, we do not describe here the positivity predicate, eliminations rules and destructors. A complete description can be found either in the Coq Reference Manual [17] or in literature [46, 13].

2.1.2 The Predicative Calculus of Inductive Constructions

The λ-calculus implemented in Coq [17] is specified by the following set of sorts, axioms and rules:

$$\begin{aligned}
\mathcal{S} &= \{\text{Prop}, \text{Type}_i \mid i \in \mathbb{N}\} \\
\mathcal{A} &= \{(\text{Prop}, \text{Type}_1), (\text{Type}_i, \text{Type}_{i+1})\} \\
\mathcal{R} &= \{(s, \text{Prop}, \text{Prop}), (\text{Prop}, \text{Type}_0, \text{Type}_0), \\
&\qquad (\text{Type}_i, \text{Type}_j, \text{Type}_k \mid i \leq k, j \leq k)\}
\end{aligned}$$

Remark that the Type_0 constant is denoted by Set in the user front end of Coq.

In the general PTS formalization we did not take in account a universe hierarchy. Following the fact that any inhabitant of an universe indexed by i is also an inhabitant of the universe indexed by $i+1$, we extend the convertibility relation to the order, denoted by $\leq_{\beta\delta}$, and inductively defined as follows:

1. if $E \vdash t_1 =_{\beta\delta} t_2$ then $E \vdash t_1 \leq_{\beta\delta} t_2$

2. if $i \leq j$ then $E \vdash \text{Type}_i \leq_{\beta\delta} \text{Type}_j$

3. for any $i \in \mathbb{N}$, $E \vdash \text{Prop} \leq_{\beta\delta} \text{Type}_i$

4. if $E \vdash T =_{\beta\delta} U$ and $E; (v : T) \vdash T_1 \leq_{\beta\delta} U_1$ then $E \vdash \forall v : T, T_1 \leq_{\beta\delta} \forall v : U, U_1$

Now, the conversion rule becomes a subtyping rule:

$$\frac{E \vdash U : s \qquad E \vdash t : T \qquad E \vdash T \leq_{\beta\delta} U}{E \vdash t : U} \text{ TERM/SUB}$$

One notes that the exact subtyping relation in Coq is $\leq_{\beta\delta\iota\zeta}$, where ι is the reduction for fixpoints and "match...with" constructions, and ζ is the reduction associated to local definitions defined as follows:

$$E \vdash \text{let } x := u \text{ in } t \rhd_\zeta t\{x/u\}$$

2.2 The original module system of Coq

In his Phd dissertation, Courant [15] has formally studied the extension of PTS by a module system. He designed a system called $\mathcal{MC}$ that stands for Module Calculus. $\mathcal{MC}$ extends PTS with modules, functors, and signatures together with specialized reductions for functor applications and module accesses. In $\mathcal{MC}$, modules are considered as anonymous second class objects. Unlike ML dialects, the base language (here PTS) makes no distinction between types and values and it has some strong properties (e.g. strong normalization of terms). Thereby, mostly all ML's module system issues do not hold. In that context, Courant has built a system that is conservative with respect to the base language, strongly normalizing and that verifies the subject reduction property. He also showed that the type inference is decidable and that his system has the principal type property.

The module system implemented in Coq by Chrząszcz [11, 12] is a restriction of $\mathcal{MC}$. In his Phd dissertation, he argues that an anonymous module calculus does not fit well when considering inductive definitions and rewriting rule definitions. Thereby, he restricts the class of module expressions to the class of paths. In that context, all complex module expressions need to be named before being used in a term. In the following, we first illustrate the basic constructions of his system, and then we give a shortened version of its formalization.

2.2.1 Basic constructions

We first give a subset of the concrete Coq syntax for fields, modules and module types:

- Fields:

FI := PARAMETER *statement.* | DEFINITION *statement* := *term.* | AXIOM *statement.*
| INDUCTIVE *statement* := *ind_ def.*
| NOTATION *abbrev* := *term* | HINT *hintkey label.*

FM := *FI* | LEMMA *statement.* PROOF. ... QED.

- Modules:

M := MODULE *label* [*: Ie*]. FM_1. FM_2. ... FM_n. END *label.*
| MODULE *label* [*: Ie*] := Me.

- Module Types:

I := MODULE TYPE *label.* FI_1. FI_2. ... FI_n. END *label.*
| MODULE TYPE *label* := Ie.

- Paths:

P := *label* | *P.label*

- Module expressions:

Me := *P*

- Module type expressions:

Ie := *P*

One remarks that: lemmas are not allowed in module types, a module is optionally sealed by a module type expression (i.e. [*: Ie*]), and module and module type expressions are restricted to paths.

Modules

A module encapsulates a list of fields. It is always a named object. For instance, take the following module in Coq syntax:

MODULE *M.*
  DEFINITION *T : Set := nat.*
  DEFINITION *e : T := O.*
  DEFINITION *op : T → T → T := plus.*
END *M.*

The module M contains three definitions: T, e, and op. If we want to project one of these definitions we use the dot notation (e.g. $M.T$).

Signatures

A signature corresponds to a type of a module. For instance, a possible type for the module M could be:

Module Type *Sig*.
  Definition $T : Set := nat$.
  Parameter $e : T$.
  Parameter $op : T \to T \to T$.
End *Sig*.

Note that this signature both hides the value of fields e and op, and makes transparent the definition of T. For instance, we can seal a copy of the module M by the signature *Sig* as follows:

Module P : *Sig* := M.

In that context, the implementation (e.g. M) given for the module P' is hidden and the only information available is the one defined in the signature. Consequently, the term $P.op\ P.e\ P.e$ is not reducible and the statement $M.e = P.e$ is not provable. We say that the signature is translucent because it contains both abstract and defined fields, and that the sealing is opaque since implementation details are not propagated.

Functors

We extend the concrete syntax to take in account functor definitions:

- Modules:

M ++= Module *label* $(X_1$:*Ie*$_1) \ldots (X_n$:*Ie*$_n)$[: *Ie*]. FM_1. FM_2. $\ldots FM_n$. End *label*.

- Module Types:

I ++= Module Type *label* $(X_1$:*Ie*$_1) \ldots (X_n$:*Ie*$_n)$. FI_1. FI_2. $\ldots FI_n$. End *label*.

- Module expressions:

Me ++= $P\ P_1 \ldots P_n$

Functors are functions from modules to modules. Analogously to modules, functors are typed by functorial signatures. For instance, the following functor takes as argument a module X of type *Sig* and returns a structure containing three fields *T*, *e*, and *op*:

```
Module F (X:Sig).
  Definition T := X.T .
  Definition e := X.e.
  Definition op := fun x y => X.op y x.
End F.
```

A type for this module could be:

```
Module Type FS (X:Sig).
  Parameter T.
  Parameter e : T.
  Parameter op : T → T → T.
End FS.
```

Functors can be applied to concrete modules:

```
Module K := F M.
```

Finally, functors are generalized to the higher-order case, which means that a functor can take a functor as argument. Take, for instance:

```
Module H (X:Sig) (Y : FS) := Y X.
```

The functor H takes a module X and a functor Y and returns the application of Y to X.

2.2.2 A shortened formalization

Syntax and Specifications

The base language on which the module system is built is the one given in Section 2.1 [5]. As Courant, Chrząszcz adopts the notion of specification. A specification is a syntactic object attached to each variable in environments, structures, and signatures. It gives all informations available about a variable. The variable declaration $v : T$ given in Section 2.1.1 is now written $v : \mathtt{Ty}(T)$ where $\mathtt{Ty}(T)$ is called an abstract specification, and the definition $v : T := t$ becomes $v : \mathtt{Eq}(t : T)$ where $\mathtt{Eq}(t : T)$ is called a manifest specification.

Following [24], the formal syntax makes the difference between labels visible from the exterior of a structure, and α-convertible binders used inside a structure. Hence, a declaration is written: $\mathtt{v} \rhd v : S$, where $\mathtt{v}$ denotes the external label, v the internal binder and S the specification.

The rules for building correct environments are split into abstract formation rules that assume correct specifications, and concrete rules that build correct specifications (abstract or manifest). We first give rules that build correct environments $E \vdash \mathrm{ok}$:

$$\textsc{Env/Empty}\ \frac{}{\vdash \mathrm{ok}} \qquad \textsc{Env/Acc}\ \frac{E \vdash \mathrm{ok} \qquad (\mathtt{v} \rhd v : S) \in E}{E \vdash v : S} \qquad \textsc{Env/Insert}\ \frac{E \vdash \mathrm{ok} \qquad E \vdash S : \mathtt{spec}}{E; (\ \mathtt{v}\ \rhd v : S) \vdash \mathrm{ok}}$$

In the above rules, the judgement $E \vdash S : \mathtt{spec}$ means that S is a correct specification in the environment E.

The syntax of specifications is the following:

$$S := \mathtt{Ty}(\psi) \mid \mathtt{Eq}(\phi : \psi)$$

$$\phi := t \mid P$$

$$\psi := T \mid I$$

$$\Xi := s \mid \mathtt{modtype}$$

We use the symbols ϕ, ψ, and Ξ to make notations uniform for both term specifications and module specifications. For term specification, they stand for term t, type T and PTS sort s, respectively. For module specification, they stand for module path P, module type I and a sort `modtype`, respectively. The new sort `modtype` is used in judgements of the form $E \vdash I : \mathtt{modtype}$, whose meaning is that the module type I is correct. The following rule scheme builds correct

[5] In his Phd dissertation, Chrząszcz also consider rewriting definitions

specifications:

$$\textsc{Ty/Form}\ \frac{E \vdash \psi : \Xi}{E \vdash \mathtt{Ty}(\psi) : \mathtt{spec}} \qquad \textsc{Eq/Form}\ \frac{E \vdash \phi : \psi}{E \vdash \mathtt{Eq}(\phi : \psi) : \mathtt{spec}} \qquad \textsc{Ty/Sat}\ \frac{E \vdash \phi : \psi \qquad E \vdash \mathtt{Ty}(\psi) : \mathtt{spec}}{E \vdash \phi : \mathtt{Ty}(\psi)}$$

$$\textsc{Eq/sat}\ \frac{E \vdash \phi \approx \phi' \qquad E \vdash \phi' : \psi \qquad E \vdash \mathtt{Eq}(\phi : \psi) : \mathtt{spec}}{E \vdash \phi' : \mathtt{Eq}(\phi : \psi)}$$

where $\approx$ denotes in one hand $=_{\beta\delta}$ when ϕ and ϕ' are terms, and in the other hand the syntactic equality when ϕ and ϕ' are module paths.

Modules and Module Types

In order to introduce the formal syntax and the typing rules for modules and module types, we give the following examples in Coq syntax together with their translation into formal syntax:

```
Module Type I.
  Parameter  T : Set.
  Parameter e : T
  Parameter  op : T -> T -> T.
End I.
```

This module type I is expressed as follows in the formal syntax:

```
Sig
  T  ▷ v1 : Ty(Set)
  e  ▷ v2 : Ty(v1)
  op ▷ v3 : Ty(v1 → v1 → v1)
End
```

We define a module that implements the module type I:

```
Module P : I.
  Definition  T : Set := nat.
  Definition  e := O.
  Definition  op := plus.
End P.
```

This module P is written as follows in formal syntax:

$$
\begin{array}{l}
\texttt{P} \rhd v_1 : \\
\quad \texttt{Ty}(\texttt{Sig} \\
\quad\quad \texttt{T} \rhd v_2 : \texttt{Ty}(Set) \\
\quad\quad \texttt{e} \rhd v_3 : \texttt{Ty}(v_1) \\
\quad\quad \texttt{op} \rhd v_4 : \texttt{Ty}(v_1 \to v_1 \to v_1) \\
\quad \texttt{End}) := \\
\quad \texttt{Struct} \\
\quad\quad \texttt{T} \rhd v_5 : \texttt{Eq}(nat : Set) := nat \\
\quad\quad \texttt{e} \rhd v_6 : \texttt{Eq}(0 : v_5) := 0 \\
\quad\quad \texttt{op} \rhd v_7 : \texttt{Eq}(plus : v_5 \to v_5 \to v_5) := plus \\
\quad \texttt{End}
\end{array}
$$

The syntax of structures and signatures is the following:

- Signatures:

$I \quad := \quad \texttt{Sig}\, \texttt{v}_1 \rhd v_1 : S_1 \ldots \texttt{v}_n \rhd v_n : S_n\, \texttt{End}$

- Structures:

$M \quad := \quad \texttt{Struct}\, \texttt{v}_1 \rhd v_1 : S_1 := m \ldots \texttt{v}_n \rhd v_n : S_n := m\, \texttt{End}$

- Implementations:

$m \quad := \quad t \,|\, M \,|\, P$

The rules used for forming correct module types and typing structures are given by the following rule scheme:

SIG/FORM

$$\frac{E, v_1 : S_1, \ldots, v_n : S_n \vdash \text{ok}}{E \vdash \texttt{Sig}\, v_1 : S_1, \ldots, v_n : S_n\, \texttt{End} : \texttt{modtype}}$$

SIG/STRUCT

$$\frac{\begin{array}{c} E \vdash \texttt{Sig}\, v_1 : S, \ldots, v_n : S_n\, \texttt{End} : \texttt{modtype} \\ \forall k \in [1 \ldots n] \qquad E, v_1 : S, \ldots, v_{k-1} : S_{k-1} \vdash m_k : S_K \end{array}}{E \vdash \texttt{Struct}\, v_1 : S_1 := m_1, \ldots, v_n : S_n := m_n\, \texttt{End} : \texttt{Sig}\, v_1 : S, \ldots, v_n : S_n\, \texttt{End}}$$

SIG/ACCES

$$\frac{E \vdash p : \texttt{Sig}\, v_1 : S_1, \ldots, v_n : S_n\, \texttt{End}}{E \vdash p.\texttt{v}_i : S_i}$$

In SIG/STRUCT, P_i represents the implementation of the field v_i. When v_i is a term field, then P_i is a term. And, when v_i is a module field then P_i is either a structure, a path, or an application of paths.

Subtyping

The subtyping relation, denoted by <:, decides if a module type is more precise than another. Given two module types I_1 and I_2, we say that I_1 is a subtype of I_2 if I_1 defines at least the same fields as I_2, and if the fields of I_1 are more (or as) precisely specified that the ones of I_2. For instance, we have:

Module Type I_1.		Module Type I_2.
Definition T : *Set* := *nat*.		Parameter T : *Set*.
Parameter e : T	<:	Parameter e : T
Parameter *op* : T -> T -> T.		End I_2.
End I_1.		

The rule for signature subtyping is defined as follows:

Sig/sub

$$\frac{\begin{array}{c} E \vdash \texttt{Sig}\, v_1 : S_1, \ldots, v_n : S_n\, \texttt{End} : \texttt{modtype} \\ E \vdash \texttt{Sig}\, v'_1 : S'_1, \ldots, v'_n : S'_m\, \texttt{End} : \texttt{modtype} \\ \{v'_1, \ldots, v'_m\} \subseteq \{v_1, \ldots, v_n\} \\ E, , v_1 : S_1, \ldots, v_n : S_n \vdash v'_k : S'_k \qquad \forall k \in [1 \ldots n] \end{array}}{E \vdash \texttt{Sig}\, v_1 : S_1, \ldots, v_n : S_n\, \texttt{End} <: \texttt{Sig}\, v'_1 : S'_1, \ldots, v'_n : S'_m\, \texttt{End}}$$

Strengthening

At this stage of the formalization, the following module definition does not type-check:

Module *P' : (I* with Definition *T:= P.T) := P.*

where *(I* with Definition *T:= P.T)* is syntactic sugar for:

Module Type *I'*.
  Definition T : *Set* := *P.T*.
  Parameter e : T
  Parameter *op* : T -> T -> T.
End *I'*.

It is explained by the fact that the type of P is not a subtype of the module type I with Definition *T:= P.T*. More formally, the following subtyping judgement does not hold:

$$\texttt{Sig } T : \texttt{Ty}(Set),\ e : \texttt{Ty}(T),\ op : \texttt{Ty}(T \to T \to T)\ \texttt{End}$$
$$<:$$
$$\texttt{Sig } T : \texttt{Eq}(P.t : Set),\ e : \texttt{Ty}(T),\ op : \texttt{Ty}(T \to T \to T)\ \texttt{End}$$

Indeed, on the right hand-side of the subtyping relation the field T is more precise than the corresponding one in the left hand-side. This is explained by the fact that we are not able to derive the principal type for a module path (in our example it concerns the type derived for the module path P). By principal type, we mean the minimal type that we can give to the path with respect of the subtyping relation. The solution is to consider the notion of strengthening [34]. This notion expresses that if a certain module path P is of type I then this path is also of type $I_{/P}$ (i.e. I strengthened by P). Where $/P$ is described as follows:

$$\begin{aligned}
\texttt{Sig } v_1 : S_1, \ldots, v_n : S_n \texttt{ End}_{/P} &= \texttt{Sig } v_{1/P.v_n} : S_1, \ldots, v_n : S_{n/P.v_n} \texttt{ End} \\
\texttt{Eq}(\phi : \psi)_{/P} &= \texttt{Eq}(\phi : \psi_{/P}) \\
\texttt{Ty}(\psi)_{/P} &= \texttt{Eq}(P : \psi_{/P}) \\
T_{/P} &= T
\end{aligned}$$

The strengthening operation recalls, to some extent, the η-expansion of a product (e.g a product t is η-expanded to $pair(fst\ t, snd\ t)$). Here each field of the signature is manifestly equal to its own projection.
Finally, we add the following rule in our typing scheme:

$$\frac{E \vdash P : I}{E \vdash P : I_{/P}}\ \textsc{str}$$

Functors

Take, the following functor definition in Coq syntax:

Module *F (X:I).*
  Definition *T := X.T .*
  Definition *e := X.e.*
  Definition *op := fun x y => X.op y x.*
End *F.*

It is expressed in formal syntax as follows:

$$F : \texttt{FunSig}(X : I)\texttt{Sig } T : \texttt{Eq}(X.T : Set),\ e : \texttt{Eq}(X.e : X.T),\ op :$$
$$\texttt{Eq}(\lambda xy \ldots : X.T \to X.T \to X.T)\ \texttt{End} :=$$
$$\texttt{Functor}(X : I)\texttt{Struct } T : \texttt{Eq}(X.T : Set) := X.T,\ e : \texttt{Eq}(X.e : X.T) := X.e,$$
$$op : \texttt{Eq}(\lambda xy \ldots : X.T \to X.T \to X.T) := \lambda xy \ldots\ \texttt{End}$$

We extend the formal syntax with `Functor` and `FunSig` constructions:

- Signatures:

$I \quad ++= \quad \texttt{FunSig}(v : I)I$

- Structures:

$M \quad ++= \quad \texttt{Functor}(v : I)M$

In order to derive well-formed functorial signatures, well-typed functors and well-typed path applications, we add the following rules in our typing scheme:

$$\textsc{Funsig/Form}\quad \frac{E \vdash I_1 : \texttt{modtype} \qquad E, v : \texttt{Ty}(I_1) \vdash I_2 : \texttt{modtype}}{E \vdash \texttt{Funsig}(v : I_1)I_2 : \texttt{modtype}}$$

$$\textsc{Funsig/Functor}\quad \frac{E, v : \texttt{Ty}(I_1) \vdash m : I_2 \qquad E \vdash \texttt{Funsig}(v : I_1)I_2 : \texttt{modtype}}{E \vdash \texttt{Functor}[v : I_1]m : \texttt{Funsig}(v : I_1)I_2}$$

$$\textsc{Mod/App}\quad \frac{E \vdash p : \texttt{Funsig}(v : I_1)I_2 \qquad E \vdash p' : I_1}{E \vdash p\,p' : I_2\{v/p'\}}$$

We also extend the subtyping relation, in the classical covariant/contravriant way, in order to consider functorial signatures:

$$\textsc{Funsig/Sub}\quad \frac{E \vdash I_1' <: I_1 \qquad E, v : \texttt{Ty}(I_1') \vdash I_2 <: I_2'}{E \vdash \texttt{Funsig}(v : I_1)I_2 <: \texttt{Funsig}(v : I_1')I_2'}$$

Finally, the strengthening on functorial signatures has no effect and hence is defined as following:

$$(\texttt{Funsig}(v : I_1)I_2)_P \quad = \quad \texttt{Funsig}(v : I_1)I_2$$

Conservativity

We give the sketch of the conservativity proof done by Chrząszcz. He argues that a closed (i.e. without axioms) Coq development corresponds to a structure typable in the empty environment. In order to simplify the conservativity proof, he assumes that the structure has the following form:

$\texttt{Struct}$

$\quad \texttt{v}_1 \rhd v_2 : S_1 := m_1 \ldots \texttt{v}_n \rhd v_n : S_n := m_n \ (1)$

$\quad \texttt{v}_{n+1} \rhd v_{n+1} : S_{n+1} := m_{n+1} \ldots \texttt{v}_k \rhd v_k : S_k := m_k \ (2)$

$\quad \texttt{v} \rhd v : \texttt{Ty}(T) := t \ (3)$

$\texttt{End}$

The part (1) is the modelling phase, it contains only term definitions and inductive type definitions. The part (2) is the proof development phase, it contains definitions of any kind and in particular module and functor definitions. The part (3) is the final theorem. Moreover, he assumes that

$$\mathtt{v}_1 \rhd v_2 : S_1 := m_1 \ldots \mathtt{v}_n \rhd v_n : S_n := m_n \vdash T : s$$

in other words, the final statement is typable in the environment of the modelling phase. He calls a such structure: a formal development, and he calls a classic formal development, a formal development where the proof development does not contain any module or functor definition.

Now, the conservativity result can be expressed as follows:

Theorem 1 *Every formal development can be transformed into a classic formal development with the same modelling phase and the same final theorem.*

The proof is built in two steps. The first step is the evaluation of module expressions into a weak head normal form. A module expression is in weak head normal form if it is a path, a structure or a functor. He defines an *eval* function that, given a formal development, evaluates iteratively from left to right each module components of the development. Once a module component is evaluated, it is given its principal signature. The second step is the flattening of module components. Given an evaluated formal development, he proceeds from right to left and eliminates the last module component. A first-order module components is replaced by its sub-components and a functorial module component is simply removed.

Conclusion

The system implemented by Chrząszcz provides modules as second class objects. Module expressions are strictly restricted to paths and functors are considered generative. This design recalls some existing ML module systems and as a matter of facts this system can be viewed as an adaptation for Coq of the system designed by Leroy in [34]. The main difference in Coq is that there is no distinction between values and types, hence a module only contains a static part. In that context, one could think that Coq and its module system support separate compilation. However, the fact that Coq implements the typical ambiguity prevents the achievement of separate compilation [14].

2.3 Motivation for an evolution of the module system

For six years, the original module system of Coq has shown its usefulness for structuring large developments, it has helped to the realization of modular libraries, like FSets, Numbers, and many user contributions. However, this system has been historically designed for ML, and is not fully adequate for Coq. In the following, we present and illustrate its limitations, and introduce our solutions to improve it.

2.3.1 An unified account for signatures and implementations of modules

Considering the statement that module types and module implementations can contain the very same objects in Coq, we claim that the classical dichotomy between them is not anymore relevant. As an illustration, we can define in the original system the following modules and module types:

Module Type *S.*	Module *M.*
Parameter *T : Set.*	Parameter *T : Set.*
Parameter *op : T -> T -> T.*	Parameter *op : T -> T -> T.*
Notation *"x ∘ y" := (op x y).*	Notation *"x ∘ y" := (op x y).*
End *S.*	End *M.*
Module Type *S'.*	Module *M'.*
Definition *T : Set := nat.*	Definition *T : Set := nat.*
Definition *op (x y : T) : T := x + y.*	Definition *op (x y : T) : T := x + y.*
Notation *"x ∘ y" := (op x y).*	Notation *"x ∘ y" := (op x y).*
End *S'.*	End *M'.*

This is explained by the fact that we have, on contrary of ML, no distinction between types and values. Hence, we now consider a unique notion of structure that unifies both module type and module implementation constructions. Furthermore, as the notion of specification has no real counterpart in the concrete implementation of the Coq module system, we abandon it and we go back to the classical notion of parameters and definitions. Now, we write $\langle \ldots \rangle$ in place of both `Sig...End` and `Struct...End`, and we call the resulting system structure-based. The structure corresponding to the module type S is now written:

$$S = \langle T : Set,\ op : T \to T \to T \rangle$$

and the module *M'* is written:

$$M' : \langle T : Set := nat,\ op : T \to T \to T := \lambda xy.(plus\, x\, y) \rangle$$

$$:= \langle T : Set := nat,\ op : T \to T \to T := \lambda xy.(plus\, x\, y) \rangle$$

We also give a higher-order structure counterpart for the constructions `Funsig` and `Functor`. Take for instance, the following functor:

```
Module F (X:S).
  Definition T := X.T .
  Definition op := X.op.
End F.
```

that is written in the structure-based system:

$$F : (X : S) \Rightarrow \langle T : Set := X.T,\ op : T \to T \to T := X.op \rangle$$

$$:= (X : S) \Rightarrow \langle T : Set := X.T,\ op : T \to T \to T := X.op \rangle$$

This new approach has two main impacts for the module system of Coq. First, it allows a simpler implementation due to a unique notion of structure. Secondly, it allows to treat uniformly the different operators on modules and module types by transferring them onto structures.

2.3.2 Higher-order functors

The original module system lacks the principal signature property for higher-order functors. For instance, take the following module definitions:

```
Module Type S.
  Parameter x : nat.
End S.

Module Type FS (X: S).
  Parameter x : nat.
End FS.

Module App (X:S)(F:FS) := F X.
```

```
Module H (X : S).
  Definition x : nat := X.x.
End H.

Module M.
  Definition x : nat := 1.
End M.

Module K := App M H.
```

The inferred signature for the module K is: `Sig` x : `Ty`(nat) `End` and do not mention that the field x is equal to 1. This is due to the fact that the system has inferred the signature `Funsig`$(X : S)$ `Funsig`$(F : FS)$ `Sig` x : `Ty`(nat) `End` for the higher-order functor *App*. Obviously, the signature of *App* is not precise enough. Indeed, it does not specify that the output of the functor *App* is the result of the application $F\ X$. This behavior makes higher-order functors totally useless.

The fully transparent signature for higher-order functors is a well-known problem for the ML community and different solutions has been proposed to solve it [35, 49, 18]. We adopt

in our new system, Leroy's applicative functor semantic [35]. We explain our choice by the following reasons. First, in our module system, the syntactic class of module expressions is strictly restricted to module paths. Hence, we are not able to apply a functor to an anonymous structure and consequently we do not loose the principal type property. Second, being pragmatic, we think that Leroy's approach is the easiest one for a concrete implementation in Coq. Finally, Coq's modules are extracted toward Ocaml's modules and this is more convenient to keep the same semantic for functors.

Adopting this semantic for our system implies that path applications are now a part of the syntactic class of module paths. Hence, we can strengthen the structure inferred for *App* by the path $F\,X$. The functor *App* has now the following structure $(X{:}S) \Rightarrow (F{:}FS) \Rightarrow \langle x : nat := (F\,X).x \rangle$ for both signature and implementation. Finally, the module K binds the following structure $\langle x : nat := (H\,M).x \rangle$ where $(H\,M).x$ reduces to 1.

2.3.3 More flexible operators on structures

Our new structure-based system allows to make operators, that usually work separately on modules and module types, uniform. These operators are module type refinement (With), module or module type inheritance (Include), and functor application. In order to treat uniformly these operators, we need to be able to extract the structure corresponding to a module type definition or a module definition. We do it as follows:

- A module type corresponds to a structure abbreviation, hence we unfold the module type definition to the corresponding structure.
- A module is defined by a path and a pair of structures where the first one stands for the type of the module and the other for its implementation. The structure corresponding to the module definition is the "type structure" strengthened by the path of the module. In other words, the structure extracted from a module corresponds to its principal type.

For instance, take the following module type and module:

```
Module Type S.
  Parameter T : Set.
  End S.

Module M : S.
  Definition T : Set := nat.
  Definition e : T := 0.
End M.
```

The extracted structure is $\langle T : Set \rangle$ for S, and $\langle T : Set := M.T \rangle$ for M.

Now, we define a new "merge of structure" operator, in order to recover both INCLUDE and WITH operators. Given two structures, this operator scans these structures comparing two fields at a time. If two fields are equally labelled, it projects in the resulting structure the most precise one. The other remaining fields, that are disjointly labelled, are simply injected in the resulting structure. To decide which field is more precise we use the subtyping relation. Concretely, a term definition is more precise than a term parameter if the type of the definition is convertible (i.e. $\leq_{\beta\delta}$) to the type of the parameter. A term definition is more precise than another definition if the type of the first one is convertible to the type of the second, and if the bodies of both are strictly convertible (i.e. $=_{\beta\delta}$). The same algorithm is performed to select module fields. However, in that case we use the structure subtyping relation. For instance, take the two following module types:

```
Module Type S.
  Parameter  T : Set.
  Definition D : nat := 0.
  Parameter F : T -> nat.
End S.

Module Type S'.
  Definition  T : Set:= bool.
  Parameter D : nat.
  Definition F (x:T) :=match x with | True => D | False => 1 end.
  Definition Z := F True.
End S'.
```

The merge of S and S' gives the structure:

$$\langle T : Set := bool,\ D : nat := 0, F : T \to nat := \lambda x : T \ldots,\ Z : nat := F\, true \rangle$$

Note that we need to prevent the merge operator from producing structures that can contain diverging definitions. For instance, the merge of the structures $\langle x : Set,\ y : set := x \rangle$ and $\langle y : Set,\ x : set := y \rangle$ needs to be rejected. This issue is well-known in mixin system [1, 28]. To solve it, one could compute the dependency graph and check if the graph resulting of a merge is acyclic. In our work, we impose the merge operator to verify if equally labeled fields of the two structures are declared in the same order. This restrictive condition implies acyclicity of the graph. However, it reduces the domain for the merge operator. For instance, $\langle x : Set,\ y : set := nat \rangle$ and $\langle y : Set,\ x : set := bool \rangle$ can not be candidate for a merging. The main advantage

is that this condition is quite simple, easily implementable, and is hence a better solution for a proof assistant.

The merge operator only works on first-order structures. However, the merge of two higher-order structures is easily done by distributivity of functor abstractions over the merge operator. Take as example the two following functors that extend a given module parameter with distinct fields:

```
Module F (X:S).
  Include X.
  Definition foo : Type := ...
End F.

Module G (X:S).
  Include X.
  Definition bar : Type := ...
End G.
```

Using + as symbol for merging, we merge the functors F and G as follows:

```
Module FG (Y:S):= (F Y)+(G Y).
```

Thanks to the strengthening, the parts of F and G, that are inherited from X, are merged as expected. The structure extracted from *(F Y)* (resp. *(G Y)*) is strengthened by the path *(F Y)* (resp. *(G Y)*), and in the context enriched by the module parameter Y, both *(F Y).T* and *(G Y).T* reduce to *Y.T*.

Lastly, this structure-based system allows a more liberal use of modules and module types since both constructions can be reified to structures. Hence a module type, which is basically a structure abbreviation, can be used to implement a module, and reciprocally a module can be used as a module type. Moreover, the merge operator is defined on structures and thus works transparently on modules or module types.

2.3.4 Dynamic namespace

A module system is a great tool to manage name-space but it lacks a kind of extensibility. By extensibility, we mean to add *a posteriori* objects in the scope of a qualifier. To illustrate, take the

List module from the standard library of Coq. Let suppose that for our personal development, we want to extend it with new functions, lemmas and notations. We have no other way than creating a module with a dummy name, like *ListExt*, importing the module *List* and then writing down our new objects. Hence, in the rest of our development we will have two distinct qualifiers *List* and *ListExt* to access objects that are of the same concern.

In this dissertation, we propose a new notion of *namespace*. It allows the user to distinguish between the name-space imposed by the modular structure of a development, and an ontological name-space that groups, under chosen qualifiers, objects of the same concern. Indeed, the name-space induced by the modular structure of a set of developments is somehow constrained by the compilation scheme. With the help of this new meta-notion of *namespace*, one can organize names in a more coherent way. We also use them to give alternative names for an object. For instance, one proves a lemma *plus_comm* that states the commutativity of the addition in the natural numbers. This lemma can inhabit different part of the name-space: in *Natural* as *plus_comm*, in *CommutativeOp* as *nat_plus_com*, in *AbelianGroups.Nat* as *commutativity*...

Considering this new notion of *namespace*, we split our module system in two parts. The first sub-system handles the name-space while the second handles structure manipulations. More concretely, the namespace system allows to define qualifiers and aliases, while the structure system manages the formation of structures by means of merging, parameterization and application. A module definition is built by the association of a local namespace and a pair of structures: one for signature, the other for implementation. In this new approach, we drop the classical distinction between internal variables and external labels: all declarations in environment or in modules are globally identified. The main advantage of a global identification of objects is that it reduces, in the concrete implementation, the amount of substitutions to compute and to apply when manipulating module objects. It is important to note that we have to take care in the formalization to not override already existing global identifiers.

2.3.5 A notion of sharing for non-logical objects

Another important task of the module system of Coq is to package together logical and non-logical objects. We call non-logical objects, all objects that are not forming a part of the base language. Typical examples are pretty-printing syntax (notations), databases for automation, or tactics. A Coq development is thus composed of a set of modules where each of them contains logical objects and non-logical objects of the same concern.

The sharing for logical object, induced by the modular structure of the development, is handled through the module system itself. Indeed, the strengthening and the WITH construction allow to add sharing constraints, while the conversion of the base language is able to solve them. However, giving a similar notion of sharing for non-logical objects has never been studied. These objects can be seen as tools, and they globally form the proof development machinery. They work outside of the Coq kernel, and they give an interface to manipulate terms and to interact with the core typing system. For obvious security issues, they manipulate an abstracted structure

of terms. Thereby, most of them do not work modulo conversion, and are not able to decide if two names actually point to the same logical object. For instance, take the following small development:

```
Module P.
  Parameter x y : T.
  Axiom foo : x=y.
  Hint Resolve foo.
End P.

Module K:=P.

Lemma bar : K.x=K.y.
Proof.
  rewrite P.foo.
```

In this example, we define a module P that contains two parameters of the same type, an axiom that states the equality of them, and a hint for the database of the AUTO tactic. Then we duplicate this module into the module K. The structure corresponding to K is:

$$\langle K.x : Type := P.x,\ K.y : Type := P.x,\ K.foo : K.x = K.y := P.foo \rangle$$

In the lemma *bar*, the tactic *rewrite* fails because it does not find, in the current goal, a sub-term that matches $P.x$ and that can be rewritten in $P.y$. Another drawback is that the hint appears twice in the database of applicable lemmas, one occurrence for $P.foo$ and the other for $K.foo$. At first sight, it does not seem so harmful, but for a big development with a substantial amount of hints, those duplications penalize proof search (i.e. it increases the branching of proof search trees). In this work, we compute an equivalence relation on names in order to derive canonical names for each logical object. Thus, every proof tool works on those canonical names, making naming issues transparent for both user and proof development machinery. In that context, the rewrite succeeds because internally the goal $K.x=K.y$ is translated to $P.x=P.y$, and the hint is not anymore duplicated since hints are now only registered for canonical names.

Even if this relation can be subsumed by the δ-reduction, we think that it is relevant to compute it. Indeed, it simplifies the work of non logical objects independently of the conversion that can be too strong or inefficient in many cases. Furthermore, the computation of this equivalence relation only relies on module derivations. Hence, the sharing problem for non-logical objects is solved independently from base language computations.

2.4 Contributions

The main contribution of this thesis is to propose an overhaul of the module system of Coq with original features to organize knowledge and to build modular developments.

Context

- We had experience feed back on the original module system from users. For instance, the generative semantic of functor is not adequate in many cases, the lack of sharing at the level of the proof writing machinery is annoying, the module system is too restrictive and the dichotomy between module and module type need to be leveled...
- Modules are second class objects, they are in complement of other vectors of modularity that are derived from inductive types.
- Modules answer the "proving-in-the-large" (e.g. organize library, name-space management...), while type classes answer the "proving-in-the-small" (ad-hoc polymorphism, automatic construction of instances...)
- Modules can package heterogeneous objects: definitions, parameters, inductive types, non-logical objects....

Realization

- We unify both module implementation and module type into a single notion of structure.
- We provide a new merge operator on structures, that subsumes both structure refinement and structure inclusion.
- We adopt an applicative semantic for functors, in the sense of the one found in OCaml.
- We formalize a new notion of namespace that allows to give alternative name scopes from the ones implied by the modular hierarchy.
- We build successively four systems and we prove the conservativity of each system with respect to its predecessor.
- We present a new equivalence relation on names, called Δ, that provides sharing for non-logical features of Coq.
- We describe the on-going implementation of our module system in the Coq proof assistant.

Chapter 3

An incremental construction of the module system

In this chapter, we give an incremental formalization of our module system. The base language is the Calculus of Constructions with namespace. Namespace gives the capacity of qualifying constant names through the dot notation. We choose to build successively four systems where each system is an extension of its predecessor. In the first one, we add the possibility to rename declarations in the environment by managing a set of alternative names for a given declaration. In the second one, we add first-order modules with the help of the structure construction. In the third one, we add a merging operator on structures. Finally, in the last system, we generalize our structure construction to the higher-order case, enabling the declaration of functors.

3.1 The base language $\mathcal{B}$

This first system extends CC_ω with namespace, we call this system $\mathcal{B}$ for base language. Here, we permit ourself to qualify names of definitions and parameters (i.e. global axioms). We distinguish, in the environment, the variables bounded in lambda abstractions or in products from global axioms.

3.1.1 Syntax and typing rules

We consider that p ranges over a set $\mathcal{P}$ called the set of qualifiers, x ranges over a set $\mathcal{X}$ called the set of identifiers, and v ranges over a set $\mathcal{V}$ called the set of variables. We add a new syntactic construction called path that corresponds to qualified name (i.e. qualifiers separated by dots).

Syntax:

- Paths:
  $P \quad := \quad Top \,|\, P.p$

- Terms:
$t, u, T, U \quad := \quad v \,|\, P.x \,|\, \lambda v : T.t \,|\, \forall v : T.U \,|\, (t\, u) \,|\, s$

- Fields:
  $e \quad := \quad (P.x : T) \,|\, (P.x : T := t)$

- Environments of declarations:
  $E \quad := \quad \epsilon \,|\, E, e \,|\, E, (v : T)$

- Namespaces:
  $\mathcal{N} \quad := \quad Top \mid \mathcal{N} :: P$

The *Top* qualifier corresponds to the initial path. The syntax of terms is quite similar to the one given in Section 2.2, except that identifiers are now qualified by paths. An environment of declarations is a list of fields and term variables. Fields are either parameters or definitions. We give a fully qualified identifier for each parameter and definition. Finally, a namespace represents the list of available paths. We call environment a pair composed by an environment of declarations and a namespace. We denote such an environment by $E \,|\, \mathcal{N}$.

Judgements $E \,|\, \mathcal{N} \vdash J$:

The system $\mathcal{B}$ defines the following judgements:

$E \,|\, \mathcal{N} \vdash \text{ok}$	The environment is well-formed.
$E \,|\, \mathcal{N} \vdash t : T$	The term t has type T.
$E \,|\, \mathcal{N} \vdash T \leq_{\beta\delta} T'$	The term T is a subtype of T'.
$E \,|\, \mathcal{N} \vdash t =_{\beta\delta} t'$	The term t is convertible to t'.

Conversion and subtyping are defined as in Chapter 2.
In the following, we use the statement $(P.x : T) \in E$ that means that E is either of the form $E_1, (P.x : T), E_2$, or $E_1, (P.x : T := t), E_2$. We use $(P.x : T) \trianglelefteq E$ (resp. $(P.x : T := t) \trianglelefteq E$) in order to denote that E is of the form $E_1, (P.x : T), E_2$ (resp. $E_1, (P.x : T := t), E_2$).

Typing rules:

- The Term class rules derive judgements of the form $E \,|\, \mathcal{N} \vdash t : T$

$$\textsc{Term/Ax}\quad \frac{E \,|\, \mathcal{N} \vdash \mathrm{ok} \qquad (s_1, s_2) \in Ax}{E \,|\, \mathcal{N} \vdash s_1 : s_2} \qquad \textsc{Term/App}\quad \frac{E \,|\, \mathcal{N} \vdash t : \forall v : U, T \qquad E \,|\, \mathcal{N} \vdash u : U}{E \,|\, \mathcal{N} \vdash (t\, u) : \{u/v\}T}$$

$$\textsc{Term/Lambda}\quad \frac{E \,|\, \mathcal{N} \vdash \forall v : T.U : s \qquad E, (v : T) \,|\, \mathcal{N} \vdash t : U}{E \,|\, \mathcal{N} \vdash \lambda v : T.t : \forall v : T.U}$$

$$\textsc{Term/Prod}\quad \frac{E \,|\, \mathcal{N} \vdash T : s_1 \qquad (s_1, s_2, s_3) \in \mathit{Prod} \qquad E, (v : T) \,|\, \mathcal{N} \vdash U : s_2}{E \,|\, \mathcal{N} \vdash \forall v : T.U : s_3}$$

$$\textsc{Term/Sub}\quad \frac{E \,|\, \mathcal{N} \vdash U : s \qquad E \,|\, \mathcal{N} \vdash t : T \qquad E \,|\, \mathcal{N} \vdash T \leq_{\beta\delta} U}{E \,|\, \mathcal{N} \vdash t : U} \qquad \textsc{Term/Acc}\quad \frac{E \,|\, \mathcal{N} \vdash \mathrm{ok} \qquad (P.x : T) \in E}{E \,|\, \mathcal{N} \vdash P.x : T}$$

$$\textsc{Term/Var}\quad \frac{E \,|\, \mathcal{N} \vdash \mathrm{ok} \qquad (v : T) \in E}{E \,|\, \mathcal{N} \vdash v : T}$$

- The Env class rules derive judgements of the form $E \,|\, \mathcal{N} \vdash \mathrm{ok}$

$$\textsc{Env/Empty}\quad \frac{}{\epsilon, \mathit{Top} \vdash \mathrm{ok}} \qquad \textsc{Env/Var}\quad \frac{E \,|\, \mathcal{N} \vdash \mathrm{ok} \qquad E \,|\, \mathcal{N} \vdash T : s \qquad s \in S}{E, (v : T) \,|\, \mathcal{N} \vdash \mathrm{ok}}$$

$$\textsc{Env/Par}\quad \frac{E \,|\, \mathcal{N} \vdash \mathrm{ok} \qquad E \,|\, \mathcal{N} \vdash T : s \qquad s \in S \qquad P \in \mathcal{N}}{E, (P.x : T) \,|\, \mathcal{N} \vdash \mathrm{ok}}$$

$$\textsc{Env/Def}\quad \frac{E \,|\, \mathcal{N} \vdash \mathrm{ok} \qquad E \,|\, \mathcal{N} \vdash t : T \qquad P \in \mathcal{N}}{E, (P.x : T := t) \,|\, \mathcal{N} \vdash \mathrm{ok}} \qquad \textsc{Env/Namespace}\quad \frac{E \,|\, \mathcal{N} \vdash \mathrm{ok} \qquad P \in \mathcal{N} \qquad P.p \notin \mathcal{N}}{E \,|\, \mathcal{N} :: P.p \vdash \mathrm{ok}}$$

This typing system is the base language for our further extensions. We denote by $E \,|\, \mathcal{N} \vdash_{\mathcal{B}} J$ a derivation in this typing system. The contribution in this system is of course the addition of the

namespace part in the environment. The namespace is here represented as a list of paths, but it can be seen as a tree labeled by qualifiers. This notion allows us to have a naming discipline and it is obvious that a such extension does not modify the expressiveness of the base language of Coq. Hence, we claim that there exits a renaming ρ that assigns qualified identifiers to fresh identifiers such that if $E \,|\, \mathcal{N} \vdash_{\mathcal{B}} J$ then $\rho(E) \vdash_{CC_\omega} \rho(J)$.

3.1.2 Adding an alternative name layer, the system $\mathcal{B}_\mathcal{I}$

In this section, we enhance system $\mathcal{B}$ so that we are able to associate a set of qualified identifiers to a given declaration in the environment. Basically, a declaration has a principal name, which is the one given in the environment of declarations, and a set of alternative names which are declared in the namespace part of the environment. We associate to each path declared in the namespace a set that maps alternative names to principal names. We call them indirection sets. We denote this new system by $\mathcal{B}_\mathcal{I}$.

Extension of the syntax:

- Namespaces:

$\mathcal{N} \;:=\; Top[\mathcal{I}] \mid \mathcal{N} :: P[\mathcal{I}]$

- Indirection sets:

$\mathcal{I} \;:=\; \epsilon \mid \mathcal{I}, (P.x \leftarrow P.x)$

Notations:

- $\mathcal{N}(P.x)$ is the qualified identifier associated to $P.x$ in the namespace $\mathcal{N}$. Either $\mathcal{N}$ is of the form $\mathcal{N}_1 :: P[\mathcal{I}_1, (P'.x' \leftarrow P.x), \mathcal{I}_2] :: \mathcal{N}_2$ and then $\mathcal{N}(P.x) = P'.x'$, or $P.x$ does not belong to the domain of $\mathcal{N}$ and then $\mathcal{N}(P.x) = P.x$.

- $\mathcal{I}_{P.x}(\mathcal{N})$ (resp. $\mathcal{I}_e(\mathcal{N})$) is the indirection set which codomain is equal to $P.x$ (resp. $P.x$ if $e \equiv (P.x : T := t)$ or $e \equiv (P.x : T)$) in the namespace $\mathcal{N}$. We denote it by $\mathcal{I}_{P.x}$ (resp. $\mathcal{I}_e$) when the namespace argument is easily inferable. We write $dom(\mathcal{I}_{P.x}(\mathcal{N}))$ the domain of the indirection set $\mathcal{I}_{P.x}(\mathcal{N})$.

- For all namespaces $\mathcal{N}$, we denote by $\mathcal{N}^*$ the namespace $\mathcal{N}$ where all indirections have been removed (i.e. $\forall P,\, \exists \mathcal{I}\ (P[\mathcal{I}] \in \mathcal{N}) \iff (P[\epsilon] \in \mathcal{N}^*)$).

- $\mathcal{N} + (P.x \leftarrow P'.x')$ denotes the addition of the indirection $(P.x \leftarrow P'.x')$ to the namespace $\mathcal{N}$. If $\mathcal{N}$ is of the form $\mathcal{N}_1 :: P'[\mathcal{I}] :: \mathcal{N}_2$, then $\mathcal{N} + (P.x \leftarrow P'.x')$ is $\mathcal{N}_1 :: P'[\mathcal{I}, (P.x \leftarrow P'.x')] :: \mathcal{N}_2$.

Extension of typing rules:

We add the following rule in the Env class rules:

$$\frac{E\,|\,\mathcal{N} \vdash \text{ok} \qquad (P.x : T) \in E \qquad P' \in N}{E\,|\,\mathcal{N} + (P.x \leftarrow P'.x') \vdash \text{ok}} \quad \textsc{Env/Indirect}$$

And we modify the rule used to access a declaration in the environment:

$$\frac{E\,|\,\mathcal{N} \vdash \text{ok} \qquad \mathcal{N}(P'.x') = P.x \qquad (P.x : T) \in E}{E\,|\,\mathcal{N} \vdash P'.x' : T} \quad \textsc{Term/Acc}$$

Finally, we adjust the δ-reduction in order to consider indirections:

$$\frac{E\,|\,\mathcal{N} \vdash \text{ok} \qquad \mathcal{N}(P'.x') = P.x \qquad (P.x : T := t) \underline{\in} E}{E\,|\,\mathcal{N} \vdash P'.x' \rhd_\delta t} \quad \textsc{Delta/Def}$$

$$\frac{E\,|\,\mathcal{N} \vdash \text{ok} \qquad \mathcal{N}(P'.x') = P.x \qquad P'.x' \neq P.x \qquad (P.x : T) \underline{\in} E}{E\,|\,\mathcal{N} \vdash P'.x' \rhd_\delta P.x} \quad \textsc{Delta/Par}$$

Remark 3.1.1 *Let $E\,|\,\mathcal{N}$ be an environment such that $(P.x : T) \in E$, we have:*

$$\forall P'.x' \in dom(\mathcal{I}_{P.x}(\mathcal{N})),\ E\,|\,\mathcal{N} \vdash P'.x' =_{\beta\delta} P.x$$

since E is of the form $E_1, (P.x : T := t), E_2$ (resp. $E_1, (P.x : T), E_2$), and $P.x$ and $P'.x'$ have t as common reduct (resp. $P.x$ is the reduct of $P'.x'$).

In this system, we add support for renaming by enhancing the namespace part of the environment with indirection sets. Now, our notion of namespace not only gives us the available qualifiers, but also gives us alternative naming views on the environment of declarations. Of course at this stage of the formalization this could be achieved by declaring a new definition in the environment of declarations for each indirection in the namespace. However, as we have seen in the Chapter 2, we want to have a naming management that works independently from the concrete development that is here represented by the environment of declarations.

Now, we show that we can safely remove one indirection from a given derivation of the system $\mathcal{B}_{\mathcal{I}}$.

Lemma 3.1.1 *Suppose that*

- E *is of the form* $E_1, (P.x : T := t), E_2$
- $\mathcal{N}$ *is of the form* $\mathcal{N}_1 :: P'[\mathcal{I}_1, (P.x \leftarrow P'.x'), \mathcal{I}_2] :: \mathcal{N}_2$

Let σ *be the substitution* $\{P'.x'/P.x\}$*, we have:*

(1) if $E \,|\, \mathcal{N} \vdash J$*, then* $E_1, (P.x : T := t), \sigma E_2 \,|\, \mathcal{N}_1 :: P'[\mathcal{I}_1, \mathcal{I}_2] :: \mathcal{N}_2 \vdash \sigma J$*.*

(2) if $E \,|\, \mathcal{N} \vdash u : U$*, then* $E \,|\, \mathcal{N} \vdash u =_{\beta\delta} \sigma u$*.*

Proof. (1) By induction on the derivation $E \,|\, \mathcal{N} \vdash J$, we inspect the last rule used in the derivation. Most of the cases are directly proved by induction hypothesis, thereby we focus on the TERM/ACC rule and the conversion.
Suppose that the last rule of the derivation is:

$$\frac{E \,|\, \mathcal{N} \vdash \text{ok} \qquad \mathcal{N}(P'.x') = P.x \qquad (P.x : T) \in E}{E \,|\, \mathcal{N} \vdash P'.x' : T}\ \textsc{Term/Acc}$$

By induction hypothesis we have $E_1, (P.x : T := t), \sigma E_2 \,|\, \mathcal{N}_1 :: P'[\mathcal{I}_1, \mathcal{I}_2] :: \mathcal{N}_2 \vdash \text{ok}$, and we know that $\mathcal{N}(P.x) = P.x$ and $(P.x : T) \in E$, thus we can use the rule TERM/ACC to derive $E_1, (P.x : T := t), \sigma E_2 \,|\, \mathcal{N}_1 :: P'[\mathcal{I}_1, \mathcal{I}_2] :: \mathcal{N}_2 \vdash P.x : T$.

Suppose that J is of the form $u =_{\beta\delta} u'$. The interesting cases are: (a) $u \equiv P'.x'$ and $u' \equiv t$, (b) $u \equiv P'.x'$ and $u' \equiv P.x$, (c) $u \equiv P'.x'$ and $u' \equiv P'.x'$. For (a), we need to prove that $E_1, (P.x : T := t), \sigma E_2 \,|\, \mathcal{N}_1 :: P'[\mathcal{I}_1, \mathcal{I}_2] :: \mathcal{N}_2 \vdash P.x =_{\beta\delta} t$ which corresponds to one δ-reduction step. For (b) and (c) we conclude by reflexivity of $=_{\beta\delta}$.

(2) By induction on the derivation of $E \,|\, \mathcal{N} \vdash u : U$, most cases are directly proved by induction, as in (1) we concentrate ourself on the rule TERM/ACC. We consider the case when the conclusion is $E \,|\, \mathcal{N} \vdash P'.x' : T$, and we conclude on $E \,|\, \mathcal{N} \vdash P'.x' =_{\beta\delta} P.x$ since $P'.x'$ and $P.x$ have t as common reduct.

□

We can prove a similar lemma if the considered field is a parameter :

Lemma 3.1.2 *Suppose that*

- E *is of the form* $E_1, (P.x : T), E_2$
- $\mathcal{N}$ *is of the form* $\mathcal{N}_1 :: P'[\mathcal{I}_1, (P.x \leftarrow P'.x'), \mathcal{I}_2] :: \mathcal{N}_2$

Let σ be the substitution $\{P'.x'/P.x\}$, we have:

(1) if $E\,|\,\mathcal{N} \vdash J$, then $E_1, (P.x : T), \sigma E_2\,|\,\mathcal{N}_1 :: P'[\mathcal{I}_1, \mathcal{I}_2] :: \mathcal{N}_2 \vdash \sigma J$.

(2) if $E\,|\,\mathcal{N} \vdash u : U$, then $E\,|\,\mathcal{N} \vdash u =_{\beta\delta} \sigma u$.

Proof. The proof is similar to the one of lemma 3.1.1. □

3.1.3 Translation from $\mathcal{B}_\mathcal{I}$ to $\mathcal{B}$

In this section, we prove that we can safely remove indirection sets from any derivations up to renaming. To do it, we first define a binary relation $\lhd_\rho$ between well-formed environments of the systems $\mathcal{B}$ and $\mathcal{B}_\mathcal{I}$. This relation is parameterized by a substitution that cumulates the indirections of the $\mathcal{B}_\mathcal{I}$ environment, and states that if two environments are in relation then their environment of declarations contains the same set of declarations.

Definition 3.1.1 *Let ρ be a substitution that maps qualified identifiers to qualified identifiers, we define inductively the binary relation $\lhd_\rho$ between environments of the systems $\mathcal{B}$ and $\mathcal{B}_\mathcal{I}$, with the following constructors:*

$$\textsc{Empty}\quad \frac{}{(\epsilon\,|\,Top)_{\mathcal{B}_\mathcal{I}} \lhd_{\{\}} (\epsilon\,|\,Top)_\mathcal{B}} \qquad \textsc{Indirect}\quad \frac{(E\,|\,\mathcal{N}) \lhd_\rho (E'\,|\,\mathcal{N}')}{(E\,|\,\mathcal{N} + (P.x \leftarrow P'.x')) \lhd_{\rho,\{P'.x'/P.x\}} (E'\,|\,\mathcal{N}')}$$

$$\textsc{Def}\quad \frac{(E\,|\,\mathcal{N}) \lhd_\rho (E'\,|\,\mathcal{N}')}{(E, (P.x : T := t)\,|\,\mathcal{N}) \lhd_\rho (E', (P.x : \rho T := \rho t)\,|\,\mathcal{N}')}$$

$$\textsc{Par}\quad \frac{(E\,|\,\mathcal{N}) \lhd_\rho (E'\,|\,\mathcal{N}')}{(E, (P.x : T)\,|\,\mathcal{N}) \lhd_\rho (E', (P.x : \rho T)\,|\,\mathcal{N}')} \qquad \textsc{Var}\quad \frac{(E\,|\,\mathcal{N}) \lhd_\rho (E'\,|\,\mathcal{N}')}{(E, (v : T)\,|\,\mathcal{N}) \lhd_\rho (E', (v : \rho T)\,|\,\mathcal{N}')}$$

$$\textsc{Namespace}\quad \frac{(E\,|\,\mathcal{N}) \lhd_\rho (E'\,|\,\mathcal{N}')}{(E\,|\,\mathcal{N} :: P) \lhd_\rho (E'\,|\,\mathcal{N}' :: P)}$$

Remark 3.1.2 *The relation $\lhd_\rho$ is functional.*

Lemma 3.1.3 *For all environments $E\,|\,\mathcal{N}$ and $E'\,|\,\mathcal{N}'$, such that $E\,|\,\mathcal{N} \lhd_\rho E'\,|\,\mathcal{N}'$ for some ρ, we have $\mathcal{N}' \equiv \mathcal{N}^*$.*

Proof. By construction of the relation, we see that it is true for the constructor EMPTY and, that the property is preserved by the constructors INDIRECT and NAMESPACE.

□

Lemma 3.1.4 *For all environment $E \,|\, \mathcal{N}$, and judgement J such that we have: a derivation of $E \,|\, \mathcal{N} \vdash_{\mathcal{B}_\mathcal{I}} J$, there exists E' and ρ such that we have*

- $(E \,|\, \mathcal{N}) \lhd_\rho (E' \,|\, \mathcal{N}^*)$
- $E' \,|\, \mathcal{N}^* \vdash_\mathcal{B} \rho J$
- *if J is of the form $t : T$ then $E \,|\, \mathcal{N} \vdash_{\mathcal{B}_\mathcal{I}} t =_\delta \rho t$*

Proof. By induction on the number of indirections declared in the namespace N. We use Lemmas 3.1.1 and 3.1.2 to build the substitution ρ and to conclude.

□

3.2 Adding structures and first-order modules, the system $\mathcal{M}$

In this system we add a new syntactic construction called structure. Basically, a structure is a list of fields associated to a sub-namespace, where fields are term declarations or module declarations. A structure is built relatively to a path, and when we associate the structure to the path we define a module. The outline of this section is the following, we first extend the system $\mathcal{B}_{\mathcal{I}}$ in order to consider structures and modules, we call this system $\mathcal{M}$. Then, we consider a rule that allows to extract the structure corresponding to a module path, and we prove that this rule is admissible in the system $\mathcal{M}$. After, we consider a system equivalent to $\mathcal{M}$ in term of expressiveness, called $\mathcal{M}_{/S}$, where modules are always declared with a fully transparent signature. Finally, we prove that modules can be "flattened" in any environment, and we give the translation from the system $\mathcal{M}_{/S}$ to the system $\mathcal{B}_{\mathcal{I}}$.

3.2.1 Extension of the syntax and the typing rules

Extension of the syntax:

- Fields:

$e \quad +\!+= \quad P : S \mid P : S := S'$

- Structures:

$S \quad := \quad \langle e_1 \dots e_n \,|\, \mathcal{N} \rangle$

In this system we have two new kinds of declarations that are module definitions ($P : S' := S$) and module parameters ($P : S$).
In the rest of this section, we use for a path $P \equiv Top.p_1 \dots p_n$ the notation $P'.P''$ in order to exhibit a prefix P' which can range from Top to $Top.p_1 \dots p_n$.

Definition 3.2.1 *Let P and P' be two paths, we define recursively the path substitution $\{P'/P\}$, abbreviated below by σ, on structures, fields, namespaces and terms as following:*

- *On paths:*

$$\sigma P'.P'' = P.P''$$

$$\sigma P'' = P'' \quad (P' \textit{ is not a prefix of } P'')$$

- *On structures:*

$$\sigma\langle e_1 \dots e_n \,|\, \mathcal{N} \rangle = \langle \sigma e_1 \dots \sigma e_n \,|\, \sigma\mathcal{N} \rangle$$

- *On fields:*

$$\sigma(P''.x_i : T := t) = \sigma P''.x_i : \sigma T := \sigma t$$

$$\sigma(P''.x_i : T) = \sigma P''.x_i : \sigma T$$

$$\sigma(P'' : S := S') = \sigma P'' : \sigma S := \sigma S'$$

$$\sigma(P'' : S) = \sigma P'' : \sigma S$$

Note that the name part of a declaration is also substituted.

- *On namespaces:*

$$\sigma(\mathcal{N} :: P''[\mathcal{I}]) = (\sigma\mathcal{N}) :: \sigma P''[\sigma\mathcal{I}]$$

$$\sigma(\mathcal{I}, (P''.x'' \leftarrow P'''.x''')) = (\sigma\mathcal{I}), (\sigma P''.x'' \leftarrow \sigma P'''.x''')$$

- *On terms:*

$$\sigma v = v$$

$$\sigma s = s$$

$$\sigma P''.x = (\sigma P'').x$$

$$\sigma\lambda v : T.t = \lambda v : \sigma T.\sigma t$$

$$\sigma\forall v : T.U = \forall v : \sigma T.\sigma U$$

$$\sigma(t\, u) = (\sigma t\, \sigma u)$$

Definition 3.2.2 *(Set of bounded qualified identifiers QI) Given ϕ that is either a structure, an environment, or a field, we denote by $QI(\phi)$ the set of bound qualified identifiers in ϕ.*

- *Qualified identifiers bounded in a structure or an environment:*

$$QI(\epsilon) = \{\,\}$$

$$QI(e_1, \ldots, e_n \,|\, \mathcal{N}) = \bigcup_{P[\mathcal{I}_p] \in \mathcal{N}} dom(\mathcal{I}_p) \bigcup_{i \in [1 \ldots n]} QI(e_i)$$

$$QI(\langle e_1 \ldots e_n \,|\, \mathcal{N} \rangle) = \bigcup_{P[\mathcal{I}_p] \in \mathcal{N}} dom(\mathcal{I}_p) \bigcup_{i \in [1 \ldots n]} QI(e_i)$$

- *Qualified identifiers bounded in a term declaration :*

$$QI(P.x : T) = \{P.x\}$$

$$QI(P.x : T := t) = \{P.x\}$$

- *Qualified identifiers bounded in a module declaration:*

$$QI(P : S) = QI(S)$$

$$QI(P : S := S') = QI(S)$$

Definition 3.2.3 *(Set of free qualified identifiers FQI_P) Given a term t, we denote by $FQI_P(t)$ the set of free qualified identifiers prefixed by the path P occurring in the term t:*

$$FQI_P(v) = \{\,\}$$

$$FQI_P(P.P'.x) = \{P.P'.x\}$$

$$FQI_P(P'.x) = \{\,\} \text{ (P is not a prefix of P')}$$

$$FQI_P(\lambda v : T.t) = FQI_P(T) \cup FQI_P(t)$$

$$FQI_P(\forall v : T, U) = FQI_P(T) \cup FQI_P(U)$$

$$FQI_P(t_1\ t_2) = FQI_P(t_1) \cup FQI_P(t_2)$$

Definition 3.2.4 *(Extension of FQI_P to structures and fields) Given a structure S (resp. a field e), we denote by $FQI_P(S)$ (resp. $FQI_P(e)$) the set of free qualified identifiers prefixed by the path P occurring in the structure S (resp. in the field e):*

$$FQI_P(\langle e_1 \ldots e_n \,|\, \mathcal{N}\rangle) = \bigcup_{i \in [1 \ldots n]} FQI_P(e_i)$$

$$FQI_P(P' : S := S') = FQI_P(S) \cup FQI_P(S')$$

$$FQI_P(P' : S) = FQI_P(S)$$

$$FQI_P(P'.x : T := t) = FQI_P(T) \cup FQI_P(t)$$

$$FQI_P(P'.x : T) = FQI_P(T)$$

We abbreviate FQI_{Top} by FQI.

New Judgements:

We define a counterpart for structures and fields of the subtyping relation defined on terms. We denote by $\subseteq$ this subtyping relation. The following new judgements are considered in the system $\mathcal{M}$:

$E \,\|\, \mathcal{N} \vdash S : \text{struct}$	The structure S is well-formed.
$E \,\|\, \mathcal{N} \vdash P : S$	The module P has type S.
$E \,\|\, \mathcal{N} \vdash S_1 \subseteq S_2$	The structure S_1 is a subtype of S_2
$E \,\|\, \mathcal{N} \vdash e_1 \subseteq e_2$	The field e_1 is a subtype of e_2

Extension of the typing rules:

We define the typing system $\mathcal{M}$ to be the system $\mathcal{B}_{\mathcal{I}}$ extended with the following set of typing rules:

- The STRUCT class rule derives judgements of the form $E \,|\, \mathcal{N} \vdash S : \text{struct}$

$$\textsc{Struct/Env}\quad \frac{E, e_1, \ldots, e_n \,|\, \mathcal{N} :: P :: \mathcal{N}_P \vdash \text{ok} \qquad \mathit{Prefix}(P, \langle e_1 \ldots e_n \,|\, \mathcal{N}_P \rangle)}{E \,|\, \mathcal{N} :: P \vdash \langle e_1 \ldots e_n \,|\, \mathcal{N}_P \rangle : \text{struct}}$$

We build a structure from a part of a well-formed environment. The *Prefix* condition means that the name of each field e_i is prefixed by P, and that all paths declared in $\mathcal{N}_p$ are also prefixed by P. More formally:

$$\mathit{Prefix}(P, \langle e_1 \ldots e_n \,|\, \mathcal{N}_P \rangle) \equiv \forall P'.x \in QI(\langle e_1 \ldots e_n \,|\, \mathcal{N}_P \rangle),\ P'.x \text{ is of the form } P.P''.x$$

We implicitly assume that the rule STRUCT/ENV captures the minimal subset $e_1, \ldots, e_n \,|\, \mathcal{N}_P$ of the environment such that $\forall P'.x \in QI(E \,|\, \mathcal{N}),\ P$ is not a prefix of P'.

- We extend the ENV class, to consider the new module declarations:

$$\textsc{Env/ModPar}\quad \frac{E \,|\, \mathcal{N} \vdash \text{ok} \qquad E \,|\, \mathcal{N} :: P \vdash S : \text{struct}}{E, (P : S) \,|\, \mathcal{N} \vdash \text{ok}}$$

$$\textsc{Env/ModDef}\quad \frac{E \,|\, \mathcal{N} \vdash \text{ok}}{E \,|\, \mathcal{N} :: P \vdash S_1 : \text{struct} \qquad E \,|\, \mathcal{N} :: P \vdash S_2 : \text{struct} \qquad E \,|\, \mathcal{N} :: P \vdash S_1 \subseteq S_2}{E, (P : S_2 := S_1) \,|\, \mathcal{N} \vdash \text{ok}}$$

We can add in the environment a module parameter or a module definition. A module definition $(P : S_2 := S_1)$ is composed by a pair of structures, where the structure S_2 acts as the signature of the module, and the structure S_1 acts as its implementation.

It is important to note that structures are derived relatively to paths, and as we have seen in the rule STRUCT/ENV, all names of fields of structures are prefixed by theirs associated paths. Hence, a module only encapsulates a namespace and a set of fields that belong to

its naming scope.

To illustrate the rules Struct/Env and Env/ModDef, we take the following interactive module defined in the empty environment and written in Coq syntax:

```
Module M.
  Parameter T : Type.
  Definition id : T -> T := fun (x:T) => x.
End M.
```

We briefly describe the derivation of the module M. We start with the well-formed empty environment, we apply the rule Env/Namespace to add the valid path $Top.M$. We add successively the parameter $Top.M.T$ and the definition $Top.M.id$ using the rules Env/Par and Env/Def respectively. Then, we apply the rule Struct/Env to encapsulate the two declarations in a structure under the path $Top.M$. Finally, we use the rule Env/ModDef to add the module in the environment. In that example, the structure that acts as the signature of M is the same as the one used for implementation.

We think that this approach mixing namespace and structure actually reflects precisely the concrete implementation of interactive modules construction in the kernel of Coq.

- The Mod class rules derive judgements of the form $E\,|\,\mathcal{N} \vdash P : S$

Mod/Acc
$$\frac{E\,|\,\mathcal{N} \vdash \text{ok} \qquad (P : S) \in E}{E\,|\,\mathcal{N} \vdash P : S}$$

Mod/Field
$$\frac{E\,|\,\mathcal{N} \vdash P : \langle e_1 \dots e_n \,|\, \mathcal{N}_P \rangle \qquad (P.P' : S) \in \langle e_1 \dots e_n \rangle}{E\,|\,\mathcal{N} \vdash P.P' : S}$$

Mod/Sub
$$\frac{E\,|\,\mathcal{N} \vdash P : S \qquad E\,|\,\mathcal{N} :: \# \vdash S^{\#} \subseteq S'^{\#}}{E\,|\,\mathcal{N} \vdash P : S'}$$

One notes that if we have $E\,|\,\mathcal{N} \vdash P : S$, then we have either:

  - P is a submodule of some module P' such that $E\,|\,\mathcal{N} \vdash P' : \langle e_1 \dots e_n \,|\, \mathcal{N}_{P'} \rangle$. We can write the path P as $P'.p_1 \dots p_n.p$, where $P'.p_1 \dots p_n$ is a valid path in $\mathcal{N}_{P'}$ and the path $P'.p_1 \dots p_n.p$ is bounded in $\langle e_1 \dots e_n \rangle$.

– P is a module bounded in the environment E, and hence $P \equiv Top.p_1 \dots p_n.p$ with $Top.p_1 \dots p_n$ a valid path of $\mathcal{N}$.

The rule MOD/SUB roughly says that we can always see a module as a less precise module. Note that since we globally identify fields, the subtyping judgment needs to be derived under a fresh path. This is the role of the symbol # which denotes an always fresh path. If S is a structure associated to the path P then $S^{\#}$ is the substituted structure $\{P/\#\}S$.

- The SUB class rules derive judgements of the form $E \,|\, \mathcal{N} \vdash S' \subseteq S$ and $E \,|\, \mathcal{N} \vdash e' \subseteq e$

SUB/STRUCT

$$\frac{E \,|\, \mathcal{N} \vdash \langle e_1 \dots e_n \,|\, \mathcal{N}_1 \rangle : \text{struct} \qquad E \,|\, \mathcal{N} \vdash \langle e'_1 \dots e'_m \,|\, \mathcal{N}_2 \rangle : \text{struct} \qquad \mathcal{N}_2 \subseteq \mathcal{N}_1 \quad \phi : [1 \dots m] \mapsto [1 \dots n] \qquad \forall i \in [1 \dots m] \qquad E, e_1, \dots, e_n \,|\, \mathcal{N} :: \mathcal{N} \vdash e_{\phi(i)} \subseteq e'_i}{E \,|\, \mathcal{N} \vdash \langle e_1 \dots e_n \,|\, \mathcal{N}_1 \rangle \subseteq \langle e'_1 \dots e'_m \,|\, \mathcal{N}_2 \rangle}$$

SUB/DEF/DEF

$$\frac{E \,|\, \mathcal{N} \vdash t =_{\beta\delta} u \qquad E \,|\, \mathcal{N} \vdash T \leq_{\beta\delta} U}{E \,|\, \mathcal{N} \vdash (P.x : T := t) \subseteq (P.x : U := u)}$$

SUB/PAR/PAR

$$\frac{E \,|\, \mathcal{N} \vdash T \leq_{\beta\delta} U}{E \,|\, \mathcal{N} \vdash (P.x : T) \subseteq (P.x : U)}$$

SUB/DEF/PAR

$$\frac{E \vdash T \leq_{\beta\delta} U}{E \,|\, \mathcal{N} \vdash (P.x : T := t) \subseteq (P.x : U)}$$

SUB/MODD/MODP

$$\frac{E \,|\, \mathcal{N} :: \# \vdash S_1^{\#} \subseteq S_3^{\#}}{E \,|\, \mathcal{N} \vdash (P : S_1 := S_2) \subseteq (P : S_3)}$$

SUB/MODD/MODD

$$\frac{E \,|\, \mathcal{N} :: \# \vdash S_1^{\#} \subseteq S_3^{\#} \qquad E \,|\, \mathcal{N} :: \# \vdash S_2^{\#} \subseteq S_4^{\#} \qquad E \,|\, \mathcal{N} :: \# \vdash S_4^{\#} \subseteq S_2^{\#}}{E \,|\, \mathcal{N} \vdash (P : S_1 := S_2) \subseteq (P : S_3 := S_4)}$$

SUB/MODP/MODP

$$\frac{E \,|\, \mathcal{N} :: \# \vdash S_1^{\#} \subseteq S_2^{\#}}{E \,|\, \mathcal{N} \vdash (P : S_1) \subseteq (P : S_2)}$$

The rules of the class SUB give an algorithm for structure subtyping. The main rule is SUB/STRUCT and in premise of this rule we have:

– Two well formed structures $\langle e_1 \dots e_n \,|\, \mathcal{N}_1 \rangle$ and $\langle e'_1 \dots e'_m \,|\, \mathcal{N}_2 \rangle$

– The namespace $\mathcal{N}_2$ is a subset of the namespace $\mathcal{N}_1$. It means that:

$$\forall P[\mathcal{I}] \in \mathcal{N}_2, \exists \mathcal{I}', P[\mathcal{I}'] \in \mathcal{N}_1 \wedge \forall (P'.x' \leftarrow P.x) \in \mathcal{I}, (P'.x' \leftarrow P.x) \in \mathcal{I}'$$

– An injective mapping ϕ that maps fields of the second structure to fields of the first one, such that their respective name parts are equal.

– m derivations of field subtyping.

The other rules Sub/*/* define the subtyping relation on fields.

- We extend the Term class, in order to consider term field projections:

  Term/Field
  $$\frac{E\,|\,\mathcal{N} \vdash P : \langle e_1 \dots e_n \,|\, \mathcal{N}_P\rangle \qquad \mathcal{N} \cup \mathcal{N}_P(P''.x) = P.P'.x \qquad (P.P'.x : T) \in \langle e_1 \dots e_n\rangle}{E\,|\,\mathcal{N} \vdash P''.x : T}$$

  The rule Term/Field says that if the qualified identifier $P''.x$ is internally (i.e. in $\mathcal{N}_P$) or globally (i.e. in $\mathcal{N}$) redirected to to the field $(P.P'.x : T)$ of the module P, then we can access to the indirection.

- Finally, we extend the δ-reduction given in Section 3.1 with:

  Delta/FieldDef
  $$\frac{E\,|\,\mathcal{N} \vdash P : \langle e_1 \dots e_n \,|\, \mathcal{N}_P\rangle \quad (\mathcal{N} \cup \mathcal{N}_P)(P''.x) = P.P'.x \qquad (P.P'.x : T := t) \trianglelefteq \langle e_1 \dots e_n\rangle}{E\,|\,\mathcal{N} \vdash P''.x \rhd_\delta t}$$

  Delta/FieldPar
  $$\frac{E\,|\,\mathcal{N} \vdash P : \langle e_1 \dots e_n \,|\, \mathcal{N}_P\rangle \quad (\mathcal{N} \cup \mathcal{N}_P)(P''.x) = P.P'.x \qquad P''.x \neq P.P'.x \qquad (P.P'.x : T) \trianglelefteq \langle e_1 \dots e_n\rangle}{E\,|\,\mathcal{N} \vdash P''.x \rhd_\delta P.P'.x}$$

We define a measure $\mathcal{D}$ on structures that correspond to the depth in term of imbricated module.

Definition 3.2.5 *Measure on structure $\mathcal{D}(S)$:*

$$\begin{aligned}
\mathcal{D}(\langle e_1 \dots e_n \,|\, \mathcal{N}\rangle) &= max(\mathcal{D}(e_1), \dots, \mathcal{D}(e_n)) \\
\mathcal{D}(P : S := S') &= 1 + \mathcal{D}(S) \\
\mathcal{D}(P : S) &= 1 + \mathcal{D}(S) \\
\mathcal{D}(P.x : T := t) &= 0 \\
\mathcal{D}(P.x : T) &= 0
\end{aligned}$$

Some basic metatheory

First, we show that an environment contains only correct declarations.

Lemma 3.2.1 *Let E be the list of declarations $e_1, \dots, e_n$. All derivations of $e_1, \dots, e_n \,|\, \mathcal{N} \vdash ok$ contain as sub-derivation $\forall i = 1 \dots n$, the derivation of $e_1, \dots, e_{i-1} \,|\, \mathcal{N}' \vdash ok$ for some $\mathcal{N}' \subseteq \mathcal{N}$ and the derivations of:*

- $e_1, \dots, e_{i-1} \,|\, \mathcal{N}' \vdash t : T$ *(resp. $T : s$) when e_i is of the form $P.x : T := t$ (resp. P.x:T).*
- $e_1, \dots, e_{i-1} \,|\, \mathcal{N}' :: P \vdash S :$ struct *when e_i is of the form $P : S$.*

- $e_1, \ldots, e_{i-1} \,|\, \mathcal{N}' :: P \vdash S : \text{struct}$, $e_1, \ldots, e_{i-1} \,|\, \mathcal{N}' :: P \vdash S' : \text{struct}$, *and* $e_1, \ldots, e_{i-1} \,|\, \mathcal{N}' :: P \vdash S' \subseteq S$ *when* e_i *is of the form* $P : S := S'$.

Proof. By induction on the derivation of $e_1, \ldots, e_n \,|\, \mathcal{N} \vdash \text{ok}$. We inspect the last rule of the derivation. For the rules Env/Def, Env/Par, Env/Var, Env/ModDef, and Env/ModPar, this is easy to prove since it is stated on the premise of each rule. For the rules Env/Namespace and Env/Indirect, the size of the namespace is smaller in the premise than in the conclusion, and hence the derivation must contain one of the previous rules.

□

Lemma 3.2.2 *For all environment* $E \,|\, \mathcal{N}$ *and judgement* J*, every derivation of* $E \,|\, \mathcal{N} \vdash J$ *contains a derivation of* $E \,|\, \mathcal{N} \vdash \text{ok}$.

Proof. By induction on the derivation. If J is ok then the lemma trivially holds. In all other rules, that do not conclude by a well-formed environment, there is at least one premise which environment is either equal to $E \,|\, \mathcal{N}$ or extends it. Thus, we can conclude by induction hypothesis, or if the environment is an extension of the environment of the conclusion by Lemma 3.2.1.

□

In the following, we state some properties on qualified identifiers appearing in terms and fields.

Lemma 3.2.3 *For all environment* $E \,|\, \mathcal{N}$*, judgement* $t : T$ *such that* $E \,|\, \mathcal{N} \vdash t : T$*, we have* $FQI(t) \subseteq QI(E \,|\, \mathcal{N})$ *and* $FQI(T) \subseteq QI(E \,|\, \mathcal{N})$.

Proof. It is straightforward by induction on the derivation of $E \,|\, \mathcal{N} \vdash t : T$.

□

Lemma 3.2.4 *For all environment* $E \,|\, \mathcal{N} :: P$*, structure* $\langle e_1 \ldots e_n \,|\, \mathcal{N}_P \rangle$ *such that* $E \,|\, \mathcal{N} :: P \vdash \langle e_1 \ldots e_n \,|\, \mathcal{N}_P \rangle : \text{struct}$*, we have for all* e_i *of the form* $(P.P'.x : T)$ *or* $(P.P'.x : T := t)$*,* $FQI(e_i) \subseteq QI(E, e_1 \ldots e_{i-1} \,|\, \mathcal{N} :: P :: \mathcal{N}_P^* + \mathcal{I}_{e_1 \ldots e_{i-1}})$.

Proof. By construction, the last rule of the derivation is Struct/Env, and we conclude by Lemmas 3.2.1 and 3.2.3.

□

We give a variant of the previous lemma for free qualified identifiers restricted to the head path of the namespace.

Lemma 3.2.5 *For all environment* $E \,|\, \mathcal{N}$ *and path* P *such that* $E \,|\, \mathcal{N} :: P \vdash \langle e_1 \ldots e_n \,|\, \mathcal{N}_P \rangle : \text{struct}$*, we have for all* e_i *of the form* $(P.P'.x : T)$ *or* $(P.P'.x : T := t)$*,* $FQI_P(e_i) \subseteq QI(e_1, \ldots, e_{i-1} \,|\, \mathcal{N}_P^* + \mathcal{I}_{e_1 \ldots e_{i-1}})$

Proof. By construction, the last rule of the derivation is Struct/Env. We know that $FQI_P(E\,|\,\mathcal{N})$ is the empty set and we conclude by Lemma 3.2.4.

□

Lemma 3.2.6 *For all environment $E\,|\,\mathcal{N}$, and path P such that $E\,|\,\mathcal{N} \vdash P : \langle e_1 \dots e_n \,|\, \mathcal{N}_P\rangle$, we have for all e_i of the form $P.P'.x : T$ or $(P.P'.x : T := t)$, we have $FQI_P(e_i) \subseteq QI(e_1 \dots e_{i-1}\,|\,{\mathcal{N}_P}^* + \mathcal{I}_{e_1 \dots e_{i-1}})$*

Proof. By induction on the depth of the module P in the environment. We show, by Lemma 3.2.1 that, there exists $E'\,|\,\mathcal{N}'$, such that $E'\,|\,\mathcal{N}' :: P \vdash \langle e_1 \dots e_n \,|\, \mathcal{N}_P\rangle$: struct and we conclude by Lemma 3.2.6.

□

We then show that we can rename a path of the namespace from an arbitrary derivation.

Lemma 3.2.7 *For all environment $E\,|\,\mathcal{N}$, judgement J, paths P and P' such that $P \in \mathcal{N}$, $P' \notin \mathcal{N}$ and $E\,|\,\mathcal{N} \vdash J$, we have $\{P/P'\}E\,|\,\{P/P'\}\mathcal{N} \vdash \{P/P'\}J$*

Proof. By induction on the derivation $E\,|\,\mathcal{N} \vdash J$. For the Term and Env class rules, we have to check that the possible path conditions on premise of the rules are still verified and we conclude by induction hypothesis. For Struct/Env, in a similar way, we remark that the prefix condition is still verified. For Mod/Sub, if $P \equiv \#$ then we substitute it by a fresh path P'.

□

Now, we state a weakening lemma for our system.

Lemma 3.2.8 *For all environments E_1, E_2, E_3, and namespaces $\mathcal{N}_1$, $\mathcal{N}_2$, and $\mathcal{N}_3$ such that $QI(E_2\,|\,\mathcal{N}_2)$ and $QI(E_3\,|\,\mathcal{N}_3)$ are disjoint, if we have $E_1, E_2\,|\,\mathcal{N}_1 \cup \mathcal{N}_2 \vdash$ ok and $E_1, E_3\,|\,\mathcal{N}_1 \cup \mathcal{N}_3 \vdash J$ for some judgement J, then we have $E_1, E_2, E_3\,|\,\mathcal{N}_1 \cup \mathcal{N}_2 \cup \mathcal{N}_3 \vdash J$.*

Proof. By induction on the derivation of $E_1, E_3\,|\,\mathcal{N}_1 \cup \mathcal{N}_3 \vdash J$. Let us look at the last rule of the derivation.

- Env class rules:
  For Env/Empty, both E_1 and E_3 are empty and we need to show that $E_2\,|\,\mathcal{N}_2 \vdash$ ok, which is an assumption of the lemma.

  For Env/Def

$$\frac{E_1, E_3\,|\,\mathcal{N}_1 \cup \mathcal{N}_3 \vdash \text{ok} \qquad E_1, E_3\,|\,\mathcal{N}_1 \cup \mathcal{N}_3 \vdash t : T \qquad P \in \mathcal{N}_1 \cup \mathcal{N}_2}{E_1, E_3, (P.x : T := t)\,|\,\mathcal{N}_1 \cup \mathcal{N}_3 \vdash \text{ok}}$$

  By applying the induction hypothesis to the premise of the rule, we have $E_1, E_2, E_3\,|\,\mathcal{N}_1 \cup \mathcal{N}_2 \cup \mathcal{N}_3 \vdash$ ok and $E_1, E_2, E_3\,|\,\mathcal{N}_1 \cup \mathcal{N}_2 \cup \mathcal{N}_3 \vdash t : T$. Since the condition on namespace

is still satisfied, we can use the rule Env/Def and we obtain $E_1, E_2, E_3, (P.x : T := t) \,|\, \mathcal{N}_1 \cup \mathcal{N}_2 \cup \mathcal{N}_3 \vdash \text{ok}$.

We can do a similar proof for the others Env/* rules.

- Term class rules:
  For Term/Acc

$$\frac{E_1, E_3 \,|\, \mathcal{N}_1 \cup \mathcal{N}_3 \vdash \text{ok} \qquad \mathcal{N}_1 \cup \mathcal{N}_3(P'.x') = P.x \qquad (P.x : T) \in E_1, E_3}{E_1, E_3 \,|\, \mathcal{N}_1 \cup \mathcal{N}_3 \vdash P'.x' : T}$$

  We need to consider two cases: we have either $(P.x : T) \in E_1$, or $(P.x : T) \in E_3$. However, in both cases we can use the induction hypothesis on the premise of the rule to get $E_1, E_2, E_3 \,|\, \mathcal{N}_1 \cup \mathcal{N}_2 \cup \mathcal{N}_3 \vdash \text{ok}$, and since the domains of each namespace are disjoint we have $\mathcal{N}_1 \cup \mathcal{N}_2 \cup \mathcal{N}_3(P'.x') = P.x$. Finally, by the rule Term/Acc we get the desired conclusion.

  The other Term/* rules are dealt by induction hypothesis.

- Mod class rules:
  The rules Mod/Acc and Mod/Field are dealt as their counterpart in the Term class. For the rule Mod/Sub:

$$\frac{E_1, E_3 \,|\, \mathcal{N}_1 \cup \mathcal{N}_3 \vdash P : S \qquad E_1, E_3 \,|\, (\mathcal{N}_1 \cup \mathcal{N}_3) :: \# \vdash S^{\#} \subseteq S'^{\#}}{E_1, E_3 \,|\, \mathcal{N}_1 \cup \mathcal{N}_3 \vdash P : S'}$$

  We apply induction hypothesis on the premise, since # is a fresh path the namespace $(\mathcal{N}_1 \cup \mathcal{N}_2 \cup \mathcal{N}_3) :: \#$ is valid. We apply the rule Mod/Sub to get the desired conclusion.

- For Struct and Sub classes, the lemma holds by induction hypothesis.

□

Transitivity of $\subseteq$

Proposition 3.2.1 *For all environment $E \,|\, \mathcal{N}$, and term declarations e, e', and e'', if $E \,|\, \mathcal{N} \vdash e \subseteq e'$ and $E \,|\, \mathcal{N} \vdash e' \subseteq e''$, then $E \,|\, \mathcal{N} \vdash e \subseteq e''$.*

Proof. We prove it by analyzing the four possible cases for the term declarations e, e', and e'', and we conclude by transitivity of $\leq_{\beta\delta}$ and $=_{\beta\delta}$.

□

Now, we state the transitivity for structure subtyping.

Lemma 3.2.9 *For all environment $E \,|\, \mathcal{N}$, and structures S_1, S_2, and S_3, if $E \,|\, \mathcal{N} \vdash S_1 \subseteq S_2$ and $E \,|\, \mathcal{N} \vdash S_2 \subseteq S_3$, then $E \,|\, \mathcal{N} \vdash S_1 \subseteq S_3$.*

Proof. Suppose that $S_1 = \langle e_1 \dots e_n \,|\, \mathcal{N}_1 \rangle$, $S_2 = \langle e'_1 \dots e'_m \,|\, \mathcal{N}_2 \rangle$, and $S_3 = \langle e''_1 \dots e''_p \,|\, \mathcal{N}_3 \rangle$. By construction, the last rule used in the derivations of $E \,|\, \mathcal{N} \vdash S_1 \subseteq S_2$ and $E \,|\, \mathcal{N} \vdash S_2 \subseteq S_3$ is Sub/Struct. The premises of the rule state that $\mathcal{N}_2 \subseteq \mathcal{N}_1$, $\mathcal{N}_3 \subseteq \mathcal{N}_2$, and:

$$\phi : [1 \dots m] \mapsto [1 \dots n] \qquad \forall i \in [1 \dots m] \qquad E, e_1, \dots, e_{\phi(i)-1} \,|\, \mathcal{N} :: {\mathcal{N}^*}_1 + \mathcal{I}_{e_1 \dots e_{\phi(i)-1}} \vdash e_{\phi(i)} \subseteq e'_i$$

$$\phi' : [1 \dots p] \mapsto [1 \dots m] \qquad \forall i \in [1 \dots p] \qquad E, e'_1, \dots, e'_{\phi'(i)-1} \,|\, \mathcal{N} :: {\mathcal{N}^*}_2 + \mathcal{I}_{e'_1 \dots e'_{\phi'(i)-1}} \vdash e'_{\phi(i)} \subseteq e''_i$$

Let $\phi'' : [1 \dots p] \mapsto [1 \dots n] = \phi' \circ \phi$, we prove by induction on the structure length p that:

$$\forall i \in [1 \dots p] \qquad E, e_1, \dots, e_{\phi''(i)-1} \,|\, \mathcal{N} :: {\mathcal{N}^*}_1 + \mathcal{I}_{e_1 \dots e_{\phi''(i)-1}} \vdash e_{\phi''(i)} \subseteq e''_i$$

We reason by cases on the form of $e_{\phi''(i)}$, $e'_{\phi'(i)}$, and e''_i. If they are term declarations, then we conclude with the Proposition 3.2.1. If they are module declarations, then we conclude by induction hypothesis.

□

Finally, we give a "weakening by subtyping" lemma.

Lemma 3.2.10 *For all environments E_1 and E_2, fields e and e', if we have*

- $E_1, e, E_2 \,|\, (\mathcal{N}_1 + \mathcal{I}_e) \cup \mathcal{N}_2 \vdash J$
- $E_1, e' \,|\, \mathcal{N}_1 + \mathcal{I}_{e'} \vdash ok$
- $E_1 \,|\, \mathcal{N}_1 \vdash e' \subseteq e$ *and* $\mathcal{I}_e \subseteq \mathcal{I}_{e'}$

then we have $E_1, e', E_2 \,|\, (\mathcal{N}_1 + \mathcal{I}_{e'}) \cup \mathcal{N}_2 \vdash J$.

Proof. By induction on the derivation of $E_1, e, E_2 \,|\, (\mathcal{N}_1 + \mathcal{I}_e) \cup \mathcal{N}_2 \vdash J$. We denote by $E \,|\, \mathcal{N}$ the environment $E_1, e, E_2 \,|\, (\mathcal{N}_1 + \mathcal{I}_e) \cup \mathcal{N}_2$, and by $E' \,|\, \mathcal{N}'$ the environment $E_1, e', E_2 \,|\, (\mathcal{N}_1 + \mathcal{I}_{e'}) \cup \mathcal{N}_2$. For the rules of the class Term, we only do the proof for Term/Acc, since the others are directly proved by induction hypothesis.

$$\frac{E \,|\, \mathcal{N} \vdash \text{ok} \qquad \mathcal{N}(P'.x') = P.x \qquad (P.x : T) \in E}{E \,|\, \mathcal{N} \vdash P'.x' : T} \text{ Term/Acc}$$

Suppose that e is of the form $(P.x : T)$ (resp. $(P.x : T := t)$), and e' is of the form $(P.x : T')$ or $(P.x : T' := t')$ (resp. $(P.x : T' := t')$). We have by induction hypothesis that $E' \,|\, \mathcal{N}' \vdash \text{ok}$, we apply the rule Term/Acc and we obtain $E' \,|\, \mathcal{N}' \vdash P'.x' : T'$ (1) since $\mathcal{I}_e \subseteq \mathcal{I}_{e'}$. We know by inversion on the hypothesis $E_1 \,|\, \mathcal{N}_1 \vdash e' \subseteq e$ that $E_1 \,|\, \mathcal{N}_1 \vdash T' \leq_{\beta\delta} T$. Thus, by Lemma 3.2.8 we have $E' \,|\, \mathcal{N}' \vdash T' \leq_{\beta\delta} T$ (2). We apply the rule Term/Sub on (1) and (2), and we get

$E' \,|\, \mathcal{N}' \vdash P'.x' : T$.

For the rule of the class Mod, we do the proof for Mod/Acc, the other are directly proved by induction hypothesis.

$$\textsc{Mod/Acc}\quad \frac{E \,|\, \mathcal{N} \vdash \text{ok} \qquad (P : S) \in E}{E \,|\, \mathcal{N} \vdash P : S}$$

Suppose that e is of the form $P : S_1 := S_2$ (resp. $P : S_1$), and e' is $P : S_1' := S_2$ (resp. $P : S_1'$ or $P : S_1' := S_3$ for some S_3 subtype of S_1'). We know by inversion on the hypothesis $E_1 \,|\, \mathcal{N}_1 \vdash e' \subseteq e$ that $E_1 \,|\, \mathcal{N}_1 :: \# \vdash {S_1'}^{\#} \subseteq S_1^{\#}$. By induction hypothesis we have $E' \,|\, \mathcal{N}' \vdash \text{ok}$ and by Lemma 3.2.8 we have $E' \,|\, \mathcal{N}' :: \# \vdash {S_1'}^{\#} \subseteq S_1^{\#}$. We apply the rule Mod/Acc and the rule Mod/Sub to get the desired conclusion.

For the conversion: reflexivity, symmetry, transitivity and Delta/Par are proved by induction hypothesis. Now, suppose that J is $P.x =_{\beta\delta} t$, e is of the form $(P.x : T := t)$, and e' is of the form $(P.x : T' := t')$. We know by inversion on the hypothesis $E_1 \,|\, \mathcal{N}_1 \vdash e' \subseteq e$ that $E_1 \,|\, \mathcal{N}_1 \vdash t' =_{\beta\delta} t$. By Lemma 3.2.8, we have $E' \,|\, \mathcal{N}' \vdash t' =_{\beta\delta} t$. By the rule Delta/Def, we have $E' \,|\, \mathcal{N}' \vdash P.x =_{\beta\delta} t'$, and hence we conclude by transitivity of $=_{\beta\delta}$.

The rules of the classes Sub and env are proved by induction hypothesis.

□

3.2.2 An admissible rule for the Struct class

In order to be able to extract a structure from a module path, we add the following rule to our system:

$$\textsc{Struct/Path}\quad \frac{E \,|\, \mathcal{N} :: P' \vdash P : S}{E \,|\, \mathcal{N} :: P' \vdash (\{P/P'\}S)_{/P' \rhd P} : \text{struct}}$$

In Struct/Path, we substitute the whole structure by $\{P/P'\}$. This substitution replaces all occurrences of the path P by P', as well in the fields as in the local namespace of the structure S. We also strengthen the structure with the path P, we use the notation $/P' \rhd P$ in order to keep a trace of the path that is used to build the considered structure. Strengthening a structure by a module path means that we enforce the term declarations, contained in the structure, to be δ-equivalent to the corresponding ones in the module. This operation is defined recursively on structures and fields as following:

A structure strengthened by P is the structure were each field is strengthened by P:

$$\langle e_1 \dots e_n \,|\, \mathcal{N} \rangle_{/P' \rhd P} = \langle e_{1/P' \rhd P} \dots e_{n/P' \rhd P} \,|\, \mathcal{N} \rangle$$

A term definition or a term parameter, identified by $P'.P''.x$ and strengthened by P, become a term definition where the body is manifestly equal to the qualified name $P.P''.x$:

$$(P'.P''.x : T := t)_{/P' \rhd P} = (P'.P''.x : T := P.P''.x)$$

$$(P'.P''.x : T)_{/P' \rhd P} = (P'.P''.x : T := P.P''.x)$$

A module definition or a module parameter, identified by $P'.P''$ and strengthened by P, become a module definition where the signature is strengthened by $P.P''$:

$$(P'.P'' : S := S')_{/P' \rhd P} = (P'.P'' : S_{/P'.P'' \rhd P.P''} := S')$$

$$(P'.P'' : S)_{/P' \rhd P} = (P'.P'' : S_{/P'.P'' \rhd P.P''} := S_{/P' \rhd P.P''})$$

In the rest of the dissertation, we use for strengthening the usual notation, $/P$, by keeping implicit the $P' \rhd$ part.

Lemma 3.2.11 *Suppose that:*

- $E \,|\, \mathcal{N} \vdash P' : S'$
- $E, (P : S) \,|\, \mathcal{N} \vdash ok$ *where* $S = (\{P/P'\}S')_{/P'}$

then we have $\forall P''.x \in QI(S)$:

$$E, (P : S) \,|\, \mathcal{N} \vdash P''.x \rhd_\delta (\{P/P'\}P'').x$$

Proof. By definition of strengthening.

□

Before proving the admissibility of the rule Struct/Path, we prove some auxiliary lemmas. They state that, if an environment contains a declaration and its strengthened version, then in every derivation, containing this environment, we can substitute the former declaration by its strengthened version.

Lemma 3.2.12 *Let suppose that:*

- $E_1 \,|\, \mathcal{N}_1 \vdash P.x : T$ *and* $E_1 \,|\, \mathcal{N}_1 \vdash T \leq_{\beta\delta} T'$

- $E_1, E_2 \,|\, \mathcal{N}_1 \cup \mathcal{N}_2 \vdash P'.x : T'$ *and* $E_1, E_2 \,|\, \mathcal{N}_1 \cup \mathcal{N}_2 \vdash P'.x \rhd_\delta P.x$

Let σ *be the substitution* $\{P.x/P'.x\}$,

$$\textit{if } E_1, E_2, E_3 \,|\, \mathcal{N}_1 \cup \mathcal{N}_2 \cup \mathcal{N}_3 \vdash J \textit{ (1), then } E_1, E_2, \sigma E_3 \,|\, \mathcal{N}_1 \cup \mathcal{N}_2 \cup \mathcal{N}_3 \vdash \sigma J \textit{ (2)}$$

Proof. We denote (1) as $E \,|\, \mathcal{N} \vdash J$ and (2) as $E' \,|\, \mathcal{N} \vdash \sigma J$. By induction on the derivation of $E \,|\, \mathcal{N} \vdash J$.
Critical cases are Term/Acc and the conversion.
For the rule Env/Acc, if the conclusion is $E \,|\, \mathcal{N} \vdash P.x : T$, then we have to prove that $E' \,|\, \mathcal{N} \vdash P'.x : T$. By induction hypothesis we have $E' \,|\, \mathcal{N} \vdash \text{ok}$, we can use Term/Acc to access the constant $P'.x$ and use Term/Sub to obtain the desired conclusion.

For the conversion, the transitivity and the symmetry of $=_\delta$ are proved by induction hypothesis. For the reflexivity, if we have $E \,|\, \mathcal{N} \vdash P.x =_\delta P.x$, then we need to show that $E' \,|\, \mathcal{N} \vdash P'.x =_\delta P'.x$, which is trivial. For Delta/Def or Delta/Par, we conclude by the hypothesis $E' \,|\, \mathcal{N} \vdash P'.x \rhd_\delta P.x$ of the lemma.

□

Lemma 3.2.13 *Let suppose that we have two sets of qualified identifiers* $P_1.x_1, \ldots P_n.x_n$ *and* $P'_1.x_1 \ldots P'_n.x_n$ *such that* $\forall i \in [1 \ldots n]$*:*

- $E_1 \,|\, \mathcal{N}_1 \vdash P_i.x_i : T_i$ *and* $E_1 \,|\, \mathcal{N}_1 \vdash T_i \leq_{\beta\delta} T'_i$
- $E_1, E_2 \,|\, \mathcal{N}_1 \cup \mathcal{N}_2 \vdash P'_i.x_i : T'_i$ *and* $E_1, E_2 \,|\, \mathcal{N}_1 \cup \mathcal{N}_2 \vdash P'_i.x_i \rhd_\delta P_i.x_i$

Let σ *be the set of substitutions* $\bigcup_{i \in [1 \ldots n]} \{P_i.x/P'_i.x\}$,

$$\textit{if } E_1, E_2, E_3 \,|\, \mathcal{N}_1 \cup \mathcal{N}_2 \cup \mathcal{N}_3 \vdash J, \textit{ then } E_1, E_2, \sigma E_3 \,|\, \mathcal{N}_1 \cup \mathcal{N}_2 \cup \mathcal{N}_3 \vdash \sigma J$$

Proof. By induction on the number of qualified identifiers. We conclude by Lemma 3.2.12.

□

Now, we prove that the rule Struct/Path is admissible. Intuitively, it means that we can perform a global structure renaming, by renaming each field of the module, one by one, and by replacing the global substitution $\{P/P'\}$ by a set of qualified identifiers substitutions.

Lemma 3.2.14 *For all environment* E*, module path* P *and namespace* $\mathcal{N} :: P'$*,*
if $E \,|\, \mathcal{N} :: P' \vdash P : \langle e_1 \ldots e_n \,|\, \mathcal{N}_P \rangle$*, then*

$$E, \{P/P'\}e_{1/P} \ldots \{P/P'\}e_{n/P} \,|\, \mathcal{N} :: P' :: \{P/P'\}\mathcal{N}_P \vdash ok$$

Proof. We denote by σ the substitution $\{P/P'\}$. We prove that $E, \sigma e_{1/P} \dots \sigma e_{n/P} \,|\, \mathcal{N} :: P' :: \sigma\mathcal{N}_P \vdash \text{ok}$ by induction (a) on the structure length i, and by induction (b) on $\mathcal{D}(\langle e_1 \dots e_n \,|\, \mathcal{N}_P \rangle)$ the depth of imbricated modules j.
For $i = 0$ and $j = 0$, we need to prove that $E \,|\, \mathcal{N} :: P' :: \sigma\mathcal{N}_p{}^* \vdash \text{ok}$. We apply the Lemma 3.2.2 on the hypothesis of the lemma, and we get $E \,|\, \mathcal{N} :: P' \vdash \text{ok}$. Now, we use the the rule ENV/NAMESPACE to add successively the valid paths of $\sigma\mathcal{N}_p{}^*$ and we conclude on $E \,|\, \mathcal{N} :: P' :: \sigma\mathcal{N}_p{}^* \vdash \text{ok}$.

Suppose that for $i = k$ and $j = 0$, the property $E, \sigma e_{1/P} \dots \sigma e_{k/P} \,|\, \mathcal{N} :: P' :: \sigma\mathcal{N}_p{}^k \vdash \text{ok}$ holds. We denote by E_k the current environment and by $\mathcal{N}'_k$ the current namespace and we prove the property for $i = k + 1$:
If e_{k+1} is a definition $P.P_{k+1}.x_{k+1} : T := t$ (resp. a parameter $P.P_{k+1}.x_{k+1} : T$), then $\sigma e_{k+1/P}$ is the definition $P'.P_{k+1}.x_{k+1} : \sigma T := P.P_{k+1}.x_{k+1}$.

We apply the Lemma 3.2.8 on the hypothesis of the lemma and we get:

$$E_k \,|\, \mathcal{N}_k \vdash P : \langle e_1 \dots e_n \,|\, \mathcal{N}_P \rangle$$

We apply the rule TERM/FIELD and we derive:

$$E_k \,|\, \mathcal{N}_k \vdash P.P_{k+1}.x_{k+1} : T$$

By Lemma 3.2.6, we have $FQI_P(T) \subseteq QI(e_1 \dots e_k \,|\, \sigma\mathcal{N}_P{}^k)$. We denote the latter set by $\overline{P.P_k.x_k}$.

By Lemma 3.2.11, we have:

$$\forall P.P_i.x_i \in \overline{P.P_k.x_k},\ E_k \,|\, \mathcal{N}_k \vdash P'.P_i.x_i \rhd_\delta P.P_i.x_i$$

We apply the Lemma 3.2.13 on the sets of qualified identifiers $\overline{P.P_k.x_k}$ and $\overline{P'.P_k.x_k}$, and on the derivation of $E_k \,|\, \mathcal{N}_k \vdash P.P_{k+1}.x_{k+1} : T$, we obtain:

$$E_k \,|\, \mathcal{N}_k \vdash P.P_{k+1}.x_{k+1} : \sigma_i T$$

where σ_i is the substitution $\{\overline{P.P_k.x_k}/\overline{P'.P_k.x_k}\}$.

Since $FQI_P(T) \subseteq QI(e_1 \dots e_k \,|\, \sigma\mathcal{N}_P{}^k)$, $\sigma_i T$ is syntacticly the same term as σT, we have $E_k \,|\, \mathcal{N}_k \vdash P.P_{k+1}.x_{k+1} : \sigma T$.

Finally, we apply the rule ENV/DEF to add the new declaration in the environment and the rule ENV/INDIRECT to add the indirections.

Suppose that for $i = k$ and $j = m$, the property $E, \sigma e_{1/P} \dots \sigma e_{k/P} \,|\, \mathcal{N} :: P' :: \sigma\mathcal{N}_p^k \vdash \text{ok}$ holds. We denote by E_k the current environment and by $\mathcal{N}'_k$ the current namespace and we prove the property for $i = k + 1$ and $j = m + 1$:
Let e_{k+1} be a module $P.P_{k+1} : \langle e_{k+1,1} \dots e_{k+1,m} \,|\, \mathcal{N}_{P.P_{k+1}} \rangle$ of depth m.

First, note that $FQI_P(e_{k+1}) \subseteq QI(e_1 \dots e_k \,|\, \mathcal{N}_P^k) \cup QI(e_{k+1})$. Indeed, e_{k+1} is a submodule of P and hence it also declares fields which name parts are prefixed by P.

From $E_k \,|\, \mathcal{N}_k \vdash \text{ok}$ we use the rule Env/Namespace to build:

$$E_k \,|\, \mathcal{N}_k :: P'.P_{k+1} \vdash \text{ok}$$

We apply the rule Mod/Field on the hypothesis of the lemma to access the module $P.P_{k+1}$, and by Lemma 3.2.8 we obtain:

$$E_k \,|\, \mathcal{N}_k :: P'.P_{k+1} \vdash P.P_{k+1} : \langle e_{k+1,1} \dots e_{k+1,m} \,|\, \mathcal{N}_{P.P_{k+1}} \rangle$$

Since $P.P_{k+1}$ is a module of depth m, we have by induction hypothesis (b):

$$E_k, \sigma' e_{k+1,1}{}_{/P.P_{k+1}}, \dots, \sigma' e_{k+1,m}{}_{/P.P_{k+1}} \,|\, \mathcal{N}_k :: P'.P_{k+1} :: \sigma' \mathcal{N}_{P.P_{k+1}} \vdash \text{ok}$$

where $\sigma' = \{P'.P_{k+1}/P.P_{k+1}\}$.

We apply the rule Struct/Env, and we get:

$$E_k \,|\, \mathcal{N}_k :: P'.P_{k+1} \vdash \langle \sigma' e_{k+1,1}{}_{/P.P_{k+1}} \dots \sigma' e_{k+1,m}{}_{/P.P_{k+1}} \,|\, \sigma' \mathcal{N}_{P.P_{k+1}} \rangle : \text{struct} \quad (1)$$

At this point, we have:

$$FQI_P(\langle \sigma' e_{k+1,1}{}_{/P.P_{k+1}} \dots \sigma' e_{k+1,m}{}_{/P.P_{k+1}} \,|\, \sigma' \mathcal{N}_{P.P_{k+1}} \rangle) = QI(e_1 \dots e_k \,|\, \mathcal{N}_P^k)$$

We apply the Lemma 3.2.13 on the two sets of qualified identifiers $\overline{P.P_k.x_k}$ and $\overline{P'.P_k.x_k}$, and on the derivation of (1), we obtain:

$$E_k \,|\, \mathcal{N}_k :: P'.P_{k+1} \vdash \langle \sigma_i \sigma' e_{k+1,1}{}_{/P.P_{k+1}} \dots \sigma_i \sigma' e_{k+1,m}{}_{/P.P_{k+1}} \,|\, \sigma_i \sigma' \mathcal{N}_{P.P_{k+1}} \rangle : \text{struct}$$

where σ_i is the substitution $\{\overline{P.P_k.x_k}/\overline{P'.P_k.x_k}\}$.

Here, the substitution $\sigma_i \sigma'$ is equivalent to the substitution σ, since we have substituted in the submodule every free qualified identifiers prefixed by P.

Finally, we apply the rule Env/ModDef to add the new declaration in the environment, and the rule Env/Indirect to add the indirections.

□

3.2.3 Removing Sealing: the system $\mathcal{M}_{/S}$

In this section we define the system $\mathcal{M}_{/S}$ to be the system $\mathcal{M}$ with the following restricted Env/ModDef rule:

$$\textsc{Env/ModDef}\quad \frac{E\,|\,\mathcal{N} \vdash \text{ok} \qquad E\,|\,\mathcal{N} :: P \vdash S : \text{struct}}{E, (P : S := S)\,|\,\mathcal{N} \vdash \text{ok}}$$

This version of Env/ModDef gives a fully transparent signature to each module definition. This alternative rule does not modify the expressiveness of the system. Indeed, thanks to the Mod/Sub rule we can postpone the sealing of the module by a more restrictive structure at access time. We define first a relation between environments of the system $\mathcal{M}$ and $\mathcal{M}_{/S}$ and then we show that if $E\,|\,\mathcal{N} \vdash_{\mathcal{M}} J$ then there exists an environment $E'\,|\,\mathcal{N}'$ in relation with $E\,|\,\mathcal{N}$ such that $E'\,|\,\mathcal{N}' \vdash_{\mathcal{M}_{/S}} J$.

Definition 3.2.6 *We define inductively the binary relation $\sqsubseteq$ between environments of the systems $\mathcal{M}_{/S}$ and $\mathcal{M}$, with the following constructors:*

$$\textsc{Empty}\quad \frac{}{(\epsilon\,|\,Top)_{\mathcal{M}} \sqsubseteq (\epsilon\,|\,Top)_{\mathcal{M}_{/S}}} \qquad \textsc{Indirect}\quad \frac{(E'\,|\,\mathcal{N}') \sqsubseteq (E\,|\,\mathcal{N})}{(E'\,|\,\mathcal{N}' + (P.x \leftarrow P'.x')) \sqsubseteq (E\,|\,\mathcal{N} + (P.x \leftarrow P'.x'))}$$

$$\textsc{Def}\quad \frac{(E'\,|\,\mathcal{N}') \sqsubseteq (E\,|\,\mathcal{N})}{(E', (P.x : T := t)\,|\,\mathcal{N}') \sqsubseteq (E, (P.x : T := t)\,|\,\mathcal{N})} \qquad \textsc{Par}\quad \frac{(E'\,|\,\mathcal{N}') \sqsubseteq (E\,|\,\mathcal{N})}{(E', (P.x : T)\,|\,\mathcal{N}') \sqsubseteq (E, (P.x : T)\,|\,\mathcal{N})}$$

$$\textsc{Var}\quad \frac{(E'\,|\,\mathcal{N}') \sqsubseteq (E\,|\,\mathcal{N})}{(E', (v : T)\,|\,\mathcal{N}') \sqsubseteq (E, (v : T)\,|\,\mathcal{N})} \qquad \textsc{Namespace}\quad \frac{(E'\,|\,\mathcal{N}') \sqsubseteq (E\,|\,\mathcal{N})}{(E'\,|\,\mathcal{N}' :: P) \sqsubseteq (E\,|\,\mathcal{N} :: P)}$$

$$\textsc{ModP}\quad \frac{(E', e'_1, \ldots, e'_n\,|\,\mathcal{N}' :: P :: \mathcal{N}'_P) \sqsubseteq (E, e_1, \ldots, e_n\,|\,\mathcal{N} :: P :: \mathcal{N}_P)}{(E', (P : \langle e'_1 \ldots e'_n\,|\,\mathcal{N}'_p\rangle)\,|\,\mathcal{N}') \sqsubseteq (E, (P : \langle e_1 \ldots e_n\,|\,\mathcal{N}_p\rangle)\,|\,\mathcal{N})}$$

$$\textsc{ModD}\quad \frac{\begin{array}{c}(E', e'_1, \ldots, e'_n\,|\,\mathcal{N}' :: P :: \mathcal{N}'_P) \sqsubseteq (E, e_1, \ldots, e_n\,|\,\mathcal{N} :: P :: \mathcal{N}_P) \\ E'\,|\,\mathcal{N}' :: P \vdash_{\mathcal{M}} S : \text{struct} \qquad E'\,|\,\mathcal{N}' :: P \vdash_{\mathcal{M}} \langle e'_1 \ldots e'_n\,|\,\mathcal{N}'_p\rangle \subseteq S\end{array}}{\begin{array}{c}(E', (P : S := \langle e'_1 \ldots e'_n\,|\,\mathcal{N}'_p\rangle)\,|\,\mathcal{N}') \\ \sqsubseteq \\ (E, (P : \langle e_1 \ldots e_n\,|\,\mathcal{N}_p\rangle := \langle e_1 \ldots e_n\,|\,\mathcal{N}_p\rangle)\,|\,\mathcal{N})\end{array}}$$

Remark 3.2.1 *The relation $\sqsubseteq$ is functional.*

Remark 3.2.2 *All derivation of the system $\mathcal{M}_{/S}$ are valid in the system $\mathcal{M}$.*

Lemma 3.2.15 *For all environments $E'\,|\mathcal{N}'$ and $E\,|\mathcal{N}$, if $(E'\,|\mathcal{N}') \sqsubseteq (E\,|\mathcal{N})$, then for all qualified identifier $P.x$ and term T and t, we have $(P.x : T) \in E' \iff (P.x : T) \in E$ or $(P.x : T := t) \in E' \iff (P.x : T := t) \in E$.*

Proof. By construction of the relation $\sqsubseteq$.

□

Lemma 3.2.16 *For all environments $E'\,|\mathcal{N}'$ and $E\,|\mathcal{N}$, path P, and structures S, S' and S'' if $(E', (P : S'' := S')\,|\mathcal{N}') \sqsubseteq (E, (P : S := S)\,|\mathcal{N})$, then $E'\,|\mathcal{N}' :: P \vdash_{\mathcal{M}} S \subseteq S'$*

Proof. By induction on $\mathcal{D}(S)$. Suppose that $S \equiv \langle e_1 \dots e_n\,|\mathcal{N}_p\rangle$ and $S' \equiv \langle e'_1 \dots e'_n\,|\mathcal{N}'_p\rangle$.

For $\mathcal{D}(S) = 0$, we know by construction of $\sqsubseteq$ that $(E', e'_1, \dots, e'_n\,|\mathcal{N}' :: P :: \mathcal{N}'_P) \sqsubseteq (E, e_1, \dots, e_n\,|\mathcal{N} :: P :: \mathcal{N}_P)$. By induction on the structure length n and with the help of the Lemma 3.2.15, we easily show that the field e_i is syntactically the same field as e'_i. Hence, we conclude by reflexivity of the relation $\subseteq$.

We suppose the property to be true for $\mathcal{D}(S) = n$ and we do the proof for $\mathcal{D}(S) = n + 1$. As in the previous case, we have that each term fields of the structures S and S' are syntactically equal, and hence for each e_i of the form $(P.P'.x : T)$ or $(P.P'.x : T := t)$, we trivially have $e_i \subseteq e'_i$. Now if the considered fields is a module fields then we conclude by main induction hypothesis since the rule Sub/Struct is applied on a smaller structures.

□

Lemma 3.2.17 *For all environment $E'\,|\mathcal{N}'$, judgement J, if $E'\,|\mathcal{N}' \vdash_{\mathcal{M}} J$, then there exists an environment $E\,|\mathcal{N}$ such that:*

- $(E'\,|\mathcal{N}') \sqsubseteq (E\,|\mathcal{N})$
- *For all $(P : S := S') \in E'\,|\mathcal{N}'$ there exists $(P : S'' := S'') \in E\,|\mathcal{N}$ such that $E'\,|\mathcal{N}' \vdash_{\mathcal{M}} S'' \subseteq S'$*
- *if J is of the form $\langle e'_1 \dots e'_n\,|\mathcal{N}'_P\rangle$: struct, then there exists a structure $\langle e_1 \dots e_n\,|\mathcal{N}_P\rangle$ such that $E\,|\mathcal{N} \vdash_{\mathcal{M}_{/S}} \langle e_1 \dots e_n\,|\mathcal{N}_P\rangle$: struct and $(E', e'_1, \dots, e'_n\,|\mathcal{N}' :: P :: \mathcal{N}'_P) \sqsubseteq (E, e_1, \dots, e_n\,|\mathcal{N} :: P :: \mathcal{N}_P)$.*
- *for all other judgement J we have $E\,|\mathcal{N} \vdash_{\mathcal{M}_{/S}} J$*

Proof. By induction on the derivation of $E' \,|\, \mathcal{N}' \vdash_{\mathcal{M}} J$.
For the rules of the class ENV, we only do the proof for the case ENV/MODDEF:

$$\frac{E' \,|\, \mathcal{N}' \vdash_{\mathcal{M}} \text{ok} \qquad E' \,|\, \mathcal{N}' :: P \vdash_{\mathcal{M}} S_1 : \text{struct} \qquad E' \,|\, \mathcal{N}' :: P \vdash_{\mathcal{M}} S_2 : \text{struct} \qquad E' \,|\, \mathcal{N}' :: P \vdash_{\mathcal{M}} S_1 \subseteq S_2}{E', (P : S_2 := S_1) \,|\, \mathcal{N}' \vdash_{\mathcal{M}} \text{ok}}$$

Suppose that $S_1 \equiv \langle e'_1 \dots e'_n \,|\, \mathcal{N}'_P \rangle$, by induction hypothesis we have that there exists $E \,|\, \mathcal{N}$, $E'' \,|\, \mathcal{N}''$ and a structure $\langle e_1 \dots e_n \,|\, \mathcal{N}_P \rangle$ such that $(E' \,|\, \mathcal{N}') \sqsubseteq (E \,|\, \mathcal{N})$, $(E'' \,|\, \mathcal{N}'') \sqsubseteq (E \,|\, \mathcal{N})$, $E \,|\, \mathcal{N} \vdash_{\mathcal{M}/S} \text{ok}$ and $E'' \,|\, \mathcal{N}'' \vdash_{\mathcal{M}/S} \langle e_1 \dots e_n \,|\, \mathcal{N}_P \rangle : \text{struct}$. Since the relation $\sqsubseteq$ is functional then we have that $(E'' \,|\, \mathcal{N}'')$ and $(E \,|\, \mathcal{N})$ are the same environment. We use the rule ENV/MODDEF of the system $\mathcal{M}_{/S}$ and we get $E, (P : \langle e_1 \dots e_n \,|\, \mathcal{N}_P \rangle := \langle e_1 \dots e_n \,|\, \mathcal{N}_P \rangle) \,|\, \mathcal{N} \vdash_{\mathcal{M}/S} \text{ok}$. By constructor MODD we have that the environments are in relation. finally, we apply the lemma 3.2.16 and we obtain $E' \,|\, \mathcal{N}' \vdash_{\mathcal{M}} \langle e_1 \dots e_n \,|\, \mathcal{N}_P \rangle \subseteq S_1$.

For the rules of the class MOD, we only do the proof for the rule MOD/ACC:

$$\frac{E' \,|\, \mathcal{N}' \vdash_{\mathcal{M}} \text{ok} \qquad (P : S) \in E'}{E' \,|\, \mathcal{N}' \vdash_{\mathcal{M}} P : S}$$

We need to consider two cases: E' is either of the form $E'_1, (P : S), E'_2$, or $E'_1, (P : S := S'), E'_2$. Both cases are dealt similarly, thus we only do the module definition case. We have by induction hypothesis that there exists $E \,|\, \mathcal{N}$ such that $(E' \,|\, \mathcal{N}') \sqsubseteq (E \,|\, \mathcal{N})$, $E \,|\, \mathcal{N} \vdash_{\mathcal{M}/S} \text{ok}$, $(P : S'' := S'')$ and $E' \,|\, \mathcal{N}' \vdash_{\mathcal{M}} S'' \subseteq S'$. Hence, we conclude by applying the rule MOD/ACC and MOD/SUB.

The rules of the classes TERM and SUB are directly proved by induction hypothesis.

□

3.2.4 Translation from $\mathcal{M}_{/S}$ to $\mathcal{B}_{\mathcal{I}}$

In this section, we show that every term judgement $(t : T)$ derived in the system $\mathcal{M}_{/S}$ can be derived in the system $\mathcal{B}_{\mathcal{I}}$. In order to prove this conservativity statement, we show that the last module of the environment can be flattened and then we build a function that, given an environment of the system $\mathcal{M}_{/S}$, flattens every module fields.

Flattening Modules

First, we show that we can remove the last module in the environment and replace it by its sub-fields and its sub-namespace. We consider derivations of the system $\mathcal{M}_{/S}$.

Lemma 3.2.18 *Let $E\,|\,\mathcal{N} \equiv E_1, (P : S := S), E_2\,|\,\mathcal{N}_1 \cup \mathcal{N}_2$, where E_2 does not contain any module declaration and S is $\langle e_1 \dots e_n\,|\,\mathcal{N}_P\rangle$. Let $E'\,|\,\mathcal{N}' \equiv E_1, e_1 \dots, e_n, E_2\,|\,(\mathcal{N}_1 :: P :: \mathcal{N}_P) \cup \mathcal{N}_2$, we have for any judgement J different from $P : S$, if $E\,|\,\mathcal{N} \vdash J$ then $E'\,|\,\mathcal{N}' \vdash J$.*

Proof. By induction on the derivation of $E\,|\,\mathcal{N} \vdash J$.

- Env class rules:
  Suppose that the last rule of the derivation is Env/ModDef. By hypothesis of the lemma, we deduce that E_2 is the empty environment, since it cannot be of the form $e'_1, \dots, e'_m, (P' : S' := S')$. Hence, we have $E_1, (P : S := S)\,|\,\mathcal{N}_1 \vdash \text{ok}$ and we need to show $E_1, e_1 \dots, e_n\,|\,\mathcal{N}_1 :: P :: \mathcal{N}_P \vdash \text{ok}$. By inversion on the premise $E_1\,|\,\mathcal{N}_1 :: P \vdash S : \text{struct}$ of Env/ModDef, we get $E_1, e_1, \dots, e_n\,|\,\mathcal{N}_1 :: P :: \mathcal{N}_p \vdash \text{ok}$.

  Suppose that the last rule of the derivation is Env/Def, then E_2 is of the form $E'_2, (P'.x : T := t)$. We apply induction hypothesis on the premise of the rule, and we get:

  - $E_1, e_1 \dots, e_n, E'_2\,|\,(\mathcal{N}_1 :: P :: \mathcal{N}_P) \cup \mathcal{N}_2 \vdash \text{ok}$
  - $E_1, e_1 \dots, e_n, E'_2\,|\,(\mathcal{N}_1 :: P :: \mathcal{N}_P) \cup \mathcal{N}_2 \vdash t : T$.

  We now apply the rule Env/Def to get $E'\,|\,\mathcal{N}' \vdash \text{ok}$.
  The others rules from the class Env are dealt similarly.

- Mod class rules:
  Suppose that the last rule of the derivation is Mod/Acc:

$$\frac{E\,|\,\mathcal{N} \vdash \text{ok} \qquad (P' : S') \in E}{E\,|\,\mathcal{N} \vdash P' : S'}$$

  We have by hypothesis of the lemma that $(P' : S') \in E_1$. We apply induction hypothesis on the premise of the rule and we use the rule Mod/Acc to derive $E'\,|\,\mathcal{N}' \vdash P' : S'$.
  Suppose that the last rule of the derivation is Mod/Field:

$$\text{Mod/Field}\quad \frac{E\,|\,\mathcal{N} \vdash P' : \langle e'_1 \dots e'_m\,|\,\mathcal{N}_{P'}\rangle \qquad (P'.P_i : S') \in \langle e'_1 \dots e'_m\rangle}{E\,|\,\mathcal{N} \vdash P'.P_i : S'}$$

  We have either $P \equiv P'$, P is a prefix of P', or P is not a prefix of P'.

  - If $P \equiv P'$, then by Lemma 3.2.2 we have $E\,|\,\mathcal{N} \vdash \text{ok}$, and by induction hypothesis we have $E'\,|\,\mathcal{N}' \vdash \text{ok}$. We know from the premise of the rule Mod/Field that $(P.P_i : S') \in \langle e_1 \dots e_n\rangle$, thus we have $(P.P_i : S') \in E'$. Finally, we use the rule Mod/Acc on $E'\,|\,\mathcal{N}' \vdash \text{ok}$ and we obtain obtain $E'\,|\,\mathcal{N}' \vdash P.P_i : S'$.

– For both cases P is a prefix of P', or P is not prefix of P', we conclude by induction hypothesis.

- TERM class rules:
  We only do the proof for the rule TERM/FIELD, the others are easily proved by induction hypothesis:

  $$\textsc{Term/Field}\quad \frac{E\,|\,\mathcal{N} \vdash P' : \langle e_1 \dots e_n \,|\, \mathcal{N}_{P'}\rangle \qquad \mathcal{N}_{P'}(P'.P_i.x') = P.P_j.x \qquad (P.P_j.x : T) \in \langle e_1 \dots e_n\rangle}{E\,|\,\mathcal{N} \vdash P'.P_i.x' : T}$$

  We have either $P \equiv P'$, P is a prefix of P', or P is not a prefix of P'.

  – If $P \equiv P'$, then by Lemma 3.2.2 we have $E\,|\,\mathcal{N} \vdash \text{ok}$, and by induction hypothesis we have $E'\,|\,\mathcal{N}' \vdash \text{ok}$. We know that $\mathcal{N}'(P.P_i.x') = P.P_j.x$ since $\mathcal{N}' \equiv \mathcal{N} :: P :: \mathcal{N}_{P'}$ and $(P.P_j.x'_i : T) \in E'$. Thus we apply the rule TERM/ACC and we obtain $E'\,|\,\mathcal{N}' \vdash P.P_i.x' : T$.

  – For both cases P is a prefix of P', or P is not prefix of P', we conclude by induction hypothesis.

□

We can do a similar lemma for module parameters.

Lemma 3.2.19 *Let $E := E_1, (P : S), E_2\,|\,\mathcal{N}_1 \cup \mathcal{N}_2$, where E_2 does not contain any module declaration and S is $\langle e_1 \dots e_n \,|\, \mathcal{N}_P\rangle$. Let $E' := E_1, e_1 \dots, e_n, E_2\,|\,(\mathcal{N}_1 :: P :: \mathcal{N}_P) \cup \mathcal{N}_2$, we have for any judgement J, $E\,|\,\mathcal{N} \vdash J$ implies $E'\,|\,\mathcal{N}' \vdash J$ (unless J is $P : S$).*

Proof. The proof is the same as for Lemma 3.2.18

□

Translation

Now, we define a function *flatten* that takes an environment of the system $\mathcal{M}_{/S}$ and returns an environment of the system $\mathcal{B}_{\mathcal{I}}$, such that if $E\,|\,\mathcal{N} \vdash_{\mathcal{M}_{/S}} J$ where J is either a judgement of the form $t : T$, $U \leq_{\beta\delta} T$, or ok, then $E\,|\,\mathcal{N} \vdash_{\mathcal{B}_{\mathcal{I}}} J$. This function operates iteratively from right to left on the environment of declarations. If the last declaration is a term declaration, then the function accumulates it in the $\mathcal{B}_{\mathcal{I}}$ environment, and if the last declaration is a module declaration, then the function flattens it. In the following, we use the symbol || to split the environment between the $\mathcal{M}_{/S}$ part and the accumulated $\mathcal{B}_{\mathcal{I}}$ environment.

For term declarations, we simply accumulate the declaration in the outputted environment:

$$(1) \qquad \mathit{flatten}(E_1, P.x : T \,||\, E_2 \,|\, \mathcal{N}) = \mathit{flatten}(E_1 \,||\, P.x : T, E_2 \,|\, \mathcal{N})$$

$$(2) \qquad \mathit{flatten}(E_1, P.x : T := t \,||\, E_2 \,|\, \mathcal{N}) = \mathit{flatten}(E_1 \,||\, P.x : T := t, E_2 \,|\, \mathcal{N})$$

For a module parameter declaration, we use the Lemma 3.2.19 to decompose the module declaration into the declarations of its sub-fields and we add its sub-namespace in the global one:

$$(3) \qquad \mathit{flatten}(E_1, P : \langle e_1 \dots e_n \,|\, \mathcal{N}_P \rangle \,||\, E_2 \,|\, \mathcal{N}) = \mathit{flatten}(E_1, e_1, \dots, e_n \,||\, E_2 \,|\, \mathcal{N} :: P :: \mathcal{N}_P)$$

For a module definition, we use the Lemma 3.2.18 to decompose the module:

$$(4) \qquad \mathit{flatten}(E_1, P : \langle e_1 \dots e_n \,|\, \mathcal{N}_P \rangle := \langle e_1 \dots e_n \,|\, \mathcal{N}_P \rangle \,||\, E_2 \,|\, \mathcal{N}) =$$
$$\mathit{flatten}(E_1, e_1, \dots, e_n \,||\, E_2 \,|\, \mathcal{N} :: P :: \mathcal{N}'_P)$$

Finally, the function terminates when the environment of declarations at left hand-side of $||$ is empty:

$$\mathit{flatten}(\epsilon \,||\, E_2 \,|\, \mathcal{N}) = E_2 \,|\, \mathcal{N}$$

This function obviously terminates. We can take as lexicographic measure the pair composed by:

- The measure $\mathcal{D}_E$ defined as follows:

$$\begin{aligned}
\mathcal{D}_E(e_1, \dots, e_n) &= \mathcal{D}_E(e_1) + \dots + \mathcal{D}_E(e_n) \\
\mathcal{D}_E(\langle e_1 \dots e_n \,|\, \mathcal{N} \rangle) &= \mathcal{D}_E(e_1) + \dots + \mathcal{D}_E(e_n)) \\
\mathcal{D}_E(P : S := S) &= 1 + \mathcal{D}_E(S) \\
\mathcal{D}_E(P : S) &= 1 + \mathcal{D}_E(S) \\
\mathcal{D}_E(P.x : T := t) &= 0 \\
\mathcal{D}_E(P.x : T) &= 0
\end{aligned}$$

  It corresponds to the sum of the depth of each module (and sub-module) defined in the environment, it decreases in rules (3), and (4), and remains constat in (1) and (2).

- The number of fields in the environment on the left of $||$, that decreases in rules (1), and (2).

Lemma 3.2.20 *For all environment $E \,|\, \mathcal{N}$, and judgement J of the form $t : T$, $U \leq_{\beta\delta} T$, or* ok, *if $E \,|\, \mathcal{N} \vdash_{\mathcal{M}/S} J$ then $\mathit{flatten}(E \,|\, \mathcal{N}) \vdash_{\mathcal{M}/S} J$.*

Proof. By induction on $\mathcal{D}_E(E)$. If $\mathcal{D}_E(E) = 0$ then it is trivial since the function *flatten* is the identity. For the induction case, we conclude by the flattening Lemmas 3.2.18 and 3.2.19.

□

Lemma 3.2.21 *For all environment $E\,|\,\mathcal{N}$, and judgement J of the form $t : T$, $U \leq_{\beta\delta} T$ or* ok*, if $E\,|\,\mathcal{N} \vdash_{\mathcal{M}_{/S}} J$ then $\mathit{flatten}(E\,|\,\mathcal{N}) \vdash_{\mathcal{B}_I} J$.*

Proof. By Lemma 3.2.20, we have $\mathit{flatten}(E\,|\,\mathcal{N}) \vdash_{\mathcal{M}_{/S}} J$. Now, we show by induction on the derivation of $\mathit{flatten}(E\,|\,\mathcal{N}) \vdash_{\mathcal{M}_{/S}} J$ that $\mathit{flatten}(E\,|\,\mathcal{N}) \vdash_{\mathcal{B}_I} J$. All cases are directly prove by construction or by induction hypothesis.

□

3.3 Adding a merge operator on structures

In this section, we extend the system $\mathcal{M}$ with a new operator on structures, called the merge operator.

3.3.1 Extension of the typing rules

We add a new judgement, $E \,|\, \mathcal{N} \vdash S_1 \uplus S_2 \rightsquigarrow S_3$, that means that in the environment $E \,|\, \mathcal{N}$ the merging of the structures S_1 and S_2 yields the structure S_3. The merge operator, denoted by $\uplus$, allows to derive well-formed structure (i.e. judgement of the form $E \,|\, \mathcal{N} \vdash S_3 : \text{struct}$). Let us define this operator by adding a new set of rules MERGE and a new STRUCT rule:

STRUCT/MERGE

$$\frac{E \,|\, \mathcal{N} :: P \vdash \langle e_1 \dots e_n \,|\, \mathcal{N}_P \rangle : \text{struct} \qquad E \,|\, \mathcal{N} :: P \vdash \langle e'_1 \dots e'_m \,|\, \mathcal{N}'_P \rangle : \text{struct} \qquad E \,|\, \mathcal{N} :: P \vdash \langle e_1 \dots e_n \,|\, \mathcal{N}_P \rangle \ \uplus \ \langle e'_1 \dots e'_m \,|\, \mathcal{N}'_P \rangle \rightsquigarrow \langle e''_1 \dots e''_k \,|\, \mathcal{N}''_P \rangle}{E \,|\, \mathcal{N} :: P \vdash \langle e''_1 \dots e''_k \,|\, \mathcal{N}''_P \rangle : \text{struct}}$$

If we have two well-formed structures and the merging of them is successful, then the result is a well-formed structure.
Now, we give the typing scheme that describe the algorithm of structure merging. It is given by the following MERGE class of rules:

MERGE/EMPTY

$$\frac{}{E \,|\, \mathcal{N} \vdash \langle \epsilon \,|\, \mathcal{N}_1 \rangle \ \uplus \ \langle \epsilon \,|\, \mathcal{N}_2 \rangle \rightsquigarrow \langle \epsilon \,|\, \mathcal{N}_1 \cup \mathcal{N}_2 \rangle}$$

MERGE/LEFT

$$\frac{PathCond(e_1 \,|\, \mathcal{I}_{e_1}, S) \qquad E, e_1 \,|\, \mathcal{N} + \mathcal{I}_{e_1} \vdash \langle e_2 \dots e_n \,|\, \mathcal{N}_1 \rangle \ \uplus \ S \rightsquigarrow \langle e''_2 \dots e''_k \,|\, \mathcal{N}_3 \rangle}{E \,|\, \mathcal{N} \vdash \langle e_1 \dots e_n \,|\, \mathcal{N}_1 + \mathcal{I}_{e_1} \rangle \ \uplus \ S \rightsquigarrow \langle e_1 \, e''_2 \dots e''_k \,|\, \mathcal{N}_3 + \mathcal{I}_{e_1} \rangle}$$

MERGE/RIGHT

$$\frac{PathCond(e'_1 \,|\, \mathcal{I}_{e'_1}, S) \qquad E, e'_1 \,|\, \mathcal{N} + \mathcal{I}_{e'_1} \vdash S \ \uplus \ \langle e'_2 \dots e'_m \,|\, \mathcal{N}_2 \rangle \rightsquigarrow \langle e''_2 \dots e''_k \,|\, \mathcal{N}_3 \rangle}{E \,|\, \mathcal{N} \vdash S \ \uplus \ \langle e'_1 \dots e'_m \,|\, \mathcal{N}_2 + \mathcal{I}_{e'_1} \rangle \rightsquigarrow \langle e'_1 \, e''_2 \dots e''_k \,|\, \mathcal{N}_3 + \mathcal{I}_{e'_1} \rangle}$$

MERGE/MATCH

$$\frac{name(e_1) = name(e'_1) \qquad \mathcal{I}_{e''_1} = \mathcal{I}_{e_1} \cup \mathcal{I}_{e'_1} \qquad E, \mathcal{N} \vdash e_1 \ \uplus \ e'_1 \rightsquigarrow e''_1 \qquad E, e''_1 \,|\, \mathcal{N} + \mathcal{I}_{e''_1} \vdash \langle e_2 \dots e_n \,|\, \mathcal{N}_1 \rangle \ \uplus \ \langle e'_2 \dots e'_m \,|\, \mathcal{N}_2 \rangle \rightsquigarrow \langle e''_2 \dots e''_k \,|\, \mathcal{N}_3 \rangle}{E \,|\, \mathcal{N} \vdash \langle e_1 \dots e_n \,|\, \mathcal{N}_1 + \mathcal{I}_{e_1} \rangle \ \uplus \ \langle e'_1 \dots e'_m \,|\, \mathcal{N}_2 + \mathcal{I}_{e'_1} \rangle \rightsquigarrow \langle e''_1 \dots e''_k \,|\, \mathcal{N}_3 + \mathcal{I}_{e''_1} \rangle}$$

The merging of two structures is done incrementally according to the fields of the structures. The $PathCond(e_1 \,|\, \mathcal{I}_{e_1}, S)$ is the boolean predicate $(QI(e_i) \cup dom(\mathcal{I}_{e_1})) \cap QI(S) = \epsilon$. In other words, the set composed by the qualified identifiers of the field e_i and all their indirections is

disjoint from the set of qualified identifiers declared in the structure S.
We extend this operator to fields. Basically, it takes two fields and projects the most precise one according to the subtyping relation.

$$\frac{E\,|\,\mathcal{N} \vdash e_2 \subseteq e_1}{E\,|\,\mathcal{N} \vdash e_1 \uplus e_2 \rightsquigarrow e_2}\ \textsc{Merge/Field/Right} \qquad \frac{E\,|\,\mathcal{N} \vdash e_1 \subseteq e_2}{E\,|\,\mathcal{N} \vdash e_1 \uplus e_2 \rightsquigarrow e_1}\ \textsc{Merge/Field/Left}$$

3.3.2 Admissibility of the merge operator

Now, we show that the rule Struct/Merge is a redundant rule of the system $\mathcal{M}$.

Lemma 3.3.1 *For all environment E and fields e and e' if $E \vdash e \uplus e' \rightsquigarrow e''$, then $E \vdash e'' \subseteq e'$ and $E \vdash e'' \subseteq e$.*

Proof. It is clear that the definition of the $+$ operator on fields coincides with the subtyping. Indeed, it only selects the most precise field between e and e' using the subtyping relation.

□

Lemma 3.3.2 *For all environment $E\,|\,\mathcal{N}$, path P, and structure $\langle e_1 \dots e_n\,|\,\mathcal{N}_P\rangle$ if $E\,|\,\mathcal{N} :: P \vdash \langle e_1 \dots e_n\,|\,\mathcal{N}_P\rangle : \mathit{struct}$, then $E, e_1, \dots e_n\,|\,\mathcal{N} :: P :: \mathcal{N}_P \vdash \mathit{ok}$.*

Proof. By induction on the derivation of $E\,|\,\mathcal{N} :: P \vdash \langle e_1 \dots e_n\,|\,\mathcal{N}_P\rangle : \text{struct}$.
If the last rule of the derivation is Struct/Env then it is trivial.
If the last rule of the derivation is Struct/Merge:

$$\frac{E\,|\,\mathcal{N} :: P \vdash \langle e'_1 \dots e'_k\,|\,\mathcal{N}'\rangle : \text{struct} \qquad E\,|\,\mathcal{N} :: P \vdash \langle e''_1 \dots e''_m\,|\,\mathcal{N}''\rangle : \text{struct} \qquad E\,|\,\mathcal{N} :: P \vdash \langle e'_1 \dots e'_k\,|\,\mathcal{N}'\rangle \uplus \langle e''_1 \dots e''_m\,|\,\mathcal{N}''\rangle \rightsquigarrow \langle e_1 \dots e_n\,|\,\mathcal{N}_P\rangle}{E\,|\,\mathcal{N} :: P \vdash \langle e_1 \dots e_n\,|\,\mathcal{N}_P\rangle : \text{struct}}\ \textsc{Struct/Merge}$$

We have by induction hypothesis that $E, e'_1 \dots e'_k\,|\,\mathcal{N} :: P :: \mathcal{N}' \vdash \text{ok}$ (1) and $E, e''_1 \dots e''_m\,|\,\mathcal{N} :: P :: \mathcal{N}'' \vdash \text{ok}$ (2). By Lemma 3.2.1, we deduce that $E\,|\,\mathcal{N} :: P \vdash \text{ok}$, and hence we are able to build $E\,|\,\mathcal{N} :: P :: \mathcal{N}^*_P \vdash \text{ok}$ by successive applications of the rule Env/Namespace.
We denote by $E_i\,|\,\mathcal{N}_i$ the environment:

$$E, e_1, \dots, e_i\,|\,\mathcal{N} :: P :: \mathcal{N}^*_P + \mathcal{I}_{e_1,\dots,e_i}$$

and we prove by induction on the derivation of $E\,|\,\mathcal{N} :: P \vdash \langle e'_1 \dots e'_k\,|\,\mathcal{N}'\rangle \uplus \langle e''_1 \dots e''_m\,|\,\mathcal{N}''\rangle \rightsquigarrow \langle e_1 \dots e_n\,|\,\mathcal{N}_P\rangle$ that for all $i \in \{1 \dots n\}$, $j \in \{0 \dots k\}$ and $l \in \{0 \dots m\}$ such that

$$E_i\,|\,\mathcal{N}_i \vdash \langle e'_{j+1} \dots e'_k\,|\,\mathcal{N}'_{j+1}\rangle \uplus \langle e''_{l+1} \dots e''_m\,|\,\mathcal{N}''_{l+1}\rangle \rightsquigarrow \langle e_{i+1} \dots e_n\,|\,\mathcal{N}^*_p + \mathcal{I}_{e_{i+1},\dots,e_n}\rangle$$

where $\mathcal{N}'_{j+1} \equiv \mathcal{N}'^* + \mathcal{I}_{e'_{j+1},\dots,e'_k}$ (resp. $\mathcal{N}''_{l+1} \equiv \mathcal{N}''^* + \mathcal{I}_{e''_{l+1},\dots,e''_m}$) , we have the three following properties:

(i) $E_i \,|\, \mathcal{N}_i \vdash \text{ok}$

(ii) $E_i, e'_{j+1}, \ldots, e'_k \,|\, \mathcal{N}_i + \mathcal{I}_{e'_{j+1},\ldots,e'_k} \vdash \text{ok}$

(iii) $E_i, e''_{l+1}, \ldots, e''_m \,|\, \mathcal{N}_i + \mathcal{I}_{e''_{l+1},\ldots,e''_m} \vdash \text{ok}$

First, we prove it for $i = 1$ in order to illustrate the proof process. In fact, we use Lemma 3.2.10 (weakening by subtyping) to simulate a MERGE/MATCH step, and Lemma 3.2.8 (weakening) to simulate a MERGE/LEFT or MERGE/RIGHT step.
By Lemma 3.2.1 applied on (1) and (2), we have $E, e'_1 \,|\, \mathcal{N} :: P :: \mathcal{N}^*_P + \mathcal{I}_{e'_1} \vdash \text{ok}$ and $E, e''_1 \,|\, \mathcal{N} :: P :: \mathcal{N}^*_P + \mathcal{I}_{e''_1} \vdash \text{ok}$. We consider the 3 following cases:

- [MERGE/MATCH]: If $name(e'_1) = name(e''_1)$ and $E \,|\, \mathcal{N} :: P :: \mathcal{N}^*_P \vdash e'_1 \uplus e''_1 \rightsquigarrow e_1$ by Lemma 3.3.1 we have $E \,|\, \mathcal{N} :: P :: \mathcal{N}^*_P \vdash e_1 \subseteq e'_1$ and $E \,|\, \mathcal{N} :: P :: \mathcal{N}^*_P \vdash e_1 \subseteq e''_1$.
  We have by construction that $\mathcal{I}_{e_1} = \mathcal{I}_{e'_1} \cup \mathcal{I}_{e''_1}$.
  We apply Lemma 3.2.10 on (1), e'_1, and e_1 (resp. (2), e''_1, and e_1), and we get:
  - (ii) for $j = 1$, $E, e_1, e'_2, \ldots, e'_n \,|\, \mathcal{N}_1 + \mathcal{I}_{e'_2,\ldots,e'_k} \vdash \text{ok}$
  - (iii) for $l = 1$, $E, e_1, e''_2 \ldots, e''_m \,|\, \mathcal{N}_1 + \mathcal{I}_{e''_2,\ldots,e''_m} \vdash \text{ok}$.

  By Lemma 3.2.1 applied on either (ii) or (iii), we get (i) $E_1 \,|\, \mathcal{N}_1 \vdash \text{ok}$.

- [MERGE/LEFT]: If $PathCond(e'_1 \,|\, \mathcal{I}_{e'_1}, \langle e''_1 \ldots e''_m \,|\, \mathcal{N}'' \rangle)$, then e_1 is e'_1.
  Hence, we get (ii) for $j = 1$ from (1), and by Lemma 3.2.1 we get (i).
  Finally, by Lemma 3.2.8 applied on (2), we get (iii) for $l = 0$:

$$E, e_1, e''_1 \ldots, e''_m \,|\, \mathcal{N} :: P :: \mathcal{N}_1 + \mathcal{I}_{e''_1,\ldots,e''_m} \vdash \text{ok}$$

- [MERGE/RIGHT]: is the symmetric of [MERGE/LEFT].

We consider (i), (ii) and (iii) true for some i, j an l in their respective domain, and we prove them for $i + 1$:

$$E_{i+1} \,|\, \mathcal{N}_{i+1} \vdash \langle e'_{j+1} \ldots e'_k \,|\, \mathcal{N}'_{j+1} \rangle \uplus \langle e''_{l+1} \ldots e''_m \,|\, \mathcal{N}''_{l+1} \rangle \rightsquigarrow \langle e_{i+2} \ldots e_n \,|\, \mathcal{N}^*_p + \mathcal{I}_{e_{i+2},\ldots,e_n} \rangle$$

- [MERGE/MATCH]: If $name(e'_{j+1}) = name(e''_{l+1})$ and $E_i \,|\, \mathcal{N}_i \vdash e'_{j+1} \uplus e''_{l+1} \rightsquigarrow e_{i+1}$.
  By Lemma 3.3.1, we have $E_i \,|\, \mathcal{N}_i \vdash e_{i+1} \subseteq e'_{j+1}$ and $E_i \,|\, \mathcal{N}_i \vdash e_{i+1} \subseteq e''_{l+1}$.
  We apply Lemma 3.2.10 on the induction hypothesis (ii), and fields e'_{j+1}, and e_{i+1} (resp.(iii), e''_{l+1}, and e_{i+1}), and we get for $i + 1$:
  - (ii) for $j = j + 1$, $E_{i+1}, e'_{j+1}, \ldots, e'_n \,|\, \mathcal{N}_1 + \mathcal{I}_{e'_{j+1},\ldots,e'_k} \vdash \text{ok}$
  - (iii) for $l = l + 1$, $E_{i+1}, e_1, e''_{l+1} \ldots, e''_m \,|\, \mathcal{N}_1 + \mathcal{I}_{e''_{l+1},\ldots,e''_m} \vdash \text{ok}$.

  By Lemma 3.2.1, we deduce the property (i) $E_{i+1} \,|\, \mathcal{N}_{i+1} \vdash \text{ok}$.

- [MERGE/LEFT]: If $PathCond(e'_{j+1} \,|\, \mathcal{I}_{e'_{j+1}}, \langle e''_{l+1} \dots e''_m \,|\, \mathcal{N}''_{l+1} \rangle)$, then e_{i+1} is e'_{j+1}.
  By induction hypothesis, we get (ii) for $i+1$ and $j = j+1$.
  By Lemma 3.2.1, we deduce (i) $E_{i+1} \,|\, \mathcal{N}_{i+1} \vdash \text{ok}$.
  Finally by Lemma 3.2.8, we get (iii) $E_{i+1}, e''_{l+1} \dots, e''_m \,|\, \mathcal{N}_{i+1} + \mathcal{I}_{e''_{l+1},\dots,e''_m} \vdash \text{ok}$

- [MERGE/RIGHT]: is the symmetric of [MERGE/LEFT].

□

3.4 Adding higher-order structures, the system $\mathcal{HM}$

In this system we generalize the structure construction to the higher-order case, we call this system $\mathcal{HM}$. A higher-order structure is a structure parameterized by module variables. Consequently, we introduce the notion of functor, which corresponds to higher-order module. As announced in the Chapter 2, we adopt an applicative semantic for functor. The remainder of this section is the following, we first present the extension of the syntax and the typing rules, then we show that δ-reduction is weakly-normalizing with respect to the applicative semantic of functors, finally we show that the system $\mathcal{HM}$ is conservative with respect to the system $\mathcal{M}$.

3.4.1 Extension of the syntax and the typing rules

Extension of the syntax

$$P \quad +\!\!+= \quad v \mid (P\,P)$$

$$S \quad +\!\!+= \quad (v : S) \Rightarrow S$$

$$E \quad +\!\!+= \quad E, (v : S)$$

The syntactic class of path is extended with path variable v, and path application $(P\,P)$. The $(v : S) \Rightarrow S$ syntactic construction corresponds to higher-order structure. Finally, environments are list of fields, term variables and module variables.

Extension of the typing rules

- We extend the Env class in order to consider module variable introduction:

$$\textsc{Env/ModVar}\quad \frac{E \,|\, \mathcal{N} :: P \vdash \text{ok} \qquad E \,|\, \mathcal{N} :: P :: v \vdash S : \text{struct}}{E, (v : S) \,|\, \mathcal{N} :: (P\,v) \vdash \text{ok}}$$

  Here, the structure S is built relatively to the path variable v, and can be either higher-order or first-order. When introducing a module variable the head path of the namespace is applied to the path variable.

- We extend the Struct class with a rule to form higher-order structure:

$$\textsc{Struct/Fun}\quad \frac{E, (v : S) \,|\, \mathcal{N} :: (P\,v) \vdash S_1 : \text{struct}}{E \,|\, \mathcal{N} :: P \vdash (v : S) \Rightarrow S_1 : \text{struct}}$$

  The rule Struct/Fun says that if we have a well-formed structure S_1 built under the path $(P\,v)$ in the environment containing the considered module variable, then we can build the

higher-order structure $(v : S) \Rightarrow S_1$ well-formed under the path P.

- We extend the MOD class in order to consider a paths application as a path:

$$\frac{E \,|\, \mathcal{N} \vdash P_1 : (v : S_1) \Rightarrow S \qquad E \vdash P_2 : S_2 \qquad E \,|\, \mathcal{N} :: \# \vdash S_2^{\#}{}_{/P_2} \subseteq S_1^{\#}}{E \,|\, \mathcal{N} \vdash (P_1\, P_2) : \{v/P_2\}S} \text{ MOD/APP}$$

  Given a higher-order module P_1 and a module P_2 that fulfills the input of P_1, we can build the path $(P_1\, P_2)$ typed by the output structure of P_1, where the path variable v is substituted by P_2. The subtyping derivation is done under a fresh path, and we strengthen the structure $S_2^{\#}$ by P_2 in order to consider the most structure type for the module path P_2.

- We extend the subtyping relation, in order to consider higher-order structures:

$$\frac{E \,|\, \mathcal{N} :: P :: v \vdash S' \subseteq S \qquad E, (v : S') \,|\, \mathcal{N} :: (P\, v) \vdash S_1 \subseteq S_1'}{E \,|\, \mathcal{N} :: P \vdash (v : S) \Rightarrow S_1 \subseteq (v : S') \Rightarrow S_1'} \text{ SUB/FUN}$$

  The subtyping relation on higher-order structure is defined in the usual contravriant/covariant way.

- Finally, we extend the strengthening on high-order structures:

$$((v : S) \Rightarrow S_1)_{/P} = (v : S) \Rightarrow S_{1/(P\, v)}$$

We define a measure on structure, that is a combination of both depth and arity of the structure.

Definition 3.4.1 *Measure on structure $\mu(S)$:*

$$\begin{aligned}
\mu((v : S) \Rightarrow S') &= 1 + \mu(S) + \mu(S') \\
\mu(\langle e_1 \ldots e_n \,|\, \mathcal{N} \rangle) &= 1 + \sum_{i=1}^{n} \mu(e_i) \\
\mu(P : S := S') &= \mu(S) \\
\mu(P : S) &= \mu(S) \\
\mu(P.x : T := t) &= 0 \\
\mu(P.x : T) &= 0
\end{aligned}$$

In order to illustrate the system $\mathcal{HM}$, take the following functor in Coq syntax:

MODULE *F (v : S).*
  DEFINITION *d := v.t .*
END *F.*

We assume that S is the abbreviation for the structure $\langle\, v.t : Type_i \mid \epsilon \,\rangle$. The structure corresponding to the signature and the implementation of the functor is in formal syntax:

$$(v : \langle\, v.t : Type_i \mid \epsilon \,\rangle) \Rightarrow \langle\, (Top.F\, v).d : Type_i := v.t \mid \epsilon \,\rangle$$

Note that the field of the body of the functor is qualified by the path application $(Top.F\, v)$. We briefly describe the derivation of the functor F. We consider a well-formed environment $E \,|\, \mathcal{N}$. We apply twice the rule ENV/NAMESPACE to insert the path $Top.F$ and the path variable v. We apply the rule TERM/AX and ENV/PAR to add the parameter $(v.t : Type_i)$ in the environment, we get:

$$E, (v.t : Type_i) \,|\, \mathcal{N} :: Top.F :: v \vdash \text{ok}$$

Now, we use the rule STRUCT/ENV, and ENV/MODVAR to add the module variable in the environment:

$$E, (v : \langle\, (v.t : Type_i) \mid \epsilon \,\rangle) \,|\, \mathcal{N} :: (Top.F\, v) \vdash \text{ok}$$

We apply the rule MOD/ACC to access the module variable v, the rule TERM/FIELD to access the parameter $v.t$, and then we apply the rule ENV/DEF to add the definition $(Top.F\, v).d : Type_i := v.t$ in the environment:

$$E, (v : \langle\, (v.t : Type_i) \mid \epsilon \,\rangle), ((Top.F\, v).d : Type_i := v.t) \,|\, \mathcal{N} :: (Top.F\, v) \vdash \text{ok}$$

We now apply the rule STRUCT/ENV and STRUCT/FUN to derive the higher-order structure given previously.

$$E \,|\, \mathcal{N} :: Top.F \vdash (v : \langle\, v.t : Type_i \mid \epsilon \,\rangle) \Rightarrow \langle\, (Top.F\, v).d : Type_i := v.t \mid \epsilon \,\rangle : \text{struct}$$

Finally, we apply the rule ENV/MODDEF to add the module definition in the environment.

Now, we illustrate the path application, take for instance the following module:

MODULE *M.*
  DEFINITION *t := nat.*
END *M.*

We want to access the definition $(Top.F\,Top.M).d$. We use the rule Mod/Acc to access both modules $Top.F$ and $Top.M$. We can apply the rule Mod/App since we have :

$$\langle\, \#.t : Type_0 := Top.M.t \mid \epsilon \,\rangle \subseteq \langle\, \#.t : Type_i \mid \epsilon \,\rangle$$

We obtain:

$$E \,|\, \mathcal{N} \vdash (Top.F\,Top.M) : \langle\, (Top.F\,Top.M).d : Type_i := Top.M.t \mid \epsilon \,\rangle$$

Note that the field d in the structure is qualified by the excepted path. Finally, we use the rule Term/Field to access the desired definition. On the contrary of most module systems, no substitutions are needed for field projections.

3.4.2 Normalization of the δ-reduction in presence of applicative functors

The applicative semantic for functor leads path applications to appear in terms. Hence, a δ-reduction step (rules Delta/FieldDef and Delta/FieldPar) can involve the typing of module path applications. In a such context, we need to show that terms are normalizing with respect to the δ-reduction. This property is useful for the translation from the system $\mathcal{HM}$ to the system $\mathcal{M}$, since it allows to remove a large part of path applications in terms. In order to prove that every well-typed term in the system $\mathcal{HM}$ is weakly normalizing, we inspire ourself from the proof done for the simply typed λ-calculus in [32].

Characterization of terms in δ normal form

We denote by NF_δ the set of term in δ normal form. We give an inductive characterization of NF_δ relatively to an environment as follows:

Sort
$$\frac{E \,|\, \mathcal{N} \vdash s \in \mathcal{S}}{E \,|\, \mathcal{N} \vdash s \in \mathrm{NF}_\delta}$$

Par
$$\frac{E \,|\, \mathcal{N} \vdash \mathrm{ok} \qquad (P.x : T) \trianglelefteq E}{E \,|\, \mathcal{N} \vdash P.x \in \mathrm{NF}_\delta}$$

Var
$$\frac{E \,|\, \mathcal{N} \vdash \mathrm{ok} \qquad (v : T) \in E}{E \,|\, \mathcal{N} \vdash v \in \mathrm{NF}_\delta}$$

Lam
$$\frac{E \,|\, \mathcal{N} \vdash T \in \mathrm{NF}_\delta \qquad E, (v : T) \,|\, \mathcal{N} \vdash t \in \mathrm{NF}_\delta}{E \,|\, \mathcal{N} \vdash \lambda v : T.t \in \mathrm{NF}_\delta}$$

Prod
$$\frac{E \,|\, \mathcal{N} \vdash T \in \mathrm{NF}_\delta \qquad E, (v : T) \,|\, \mathcal{N} \vdash U \in \mathrm{NF}_\delta}{E \,|\, \mathcal{N} \vdash \forall v : T.U \in \mathrm{NF}_\delta}$$

App
$$\frac{E \,|\, \mathcal{N} \vdash t \in \mathrm{NF}_\delta \qquad E \,|\, \mathcal{N} \vdash u \in \mathrm{NF}_\delta}{E \,|\, \mathcal{N} \vdash (t\,u) \in \mathrm{NF}_\delta}$$

MPar
$$\frac{E \,|\, \mathcal{N} \vdash P : \langle e_1 \ldots e_n \,|\, \mathcal{N}_P \rangle \qquad (P.P'.x : T) \trianglelefteq \langle e_1 \ldots e_n \rangle}{E \,|\, \mathcal{N} \vdash P.P'.x \in \mathrm{NF}_\delta}$$

We extend NF_δ with modules and fields. Term parameters always belong to NF_δ, term definitions belong to NF_δ if and only if their body belongs to NF_δ. Finally, modules belongs to NF_δ if and only if all fields of the module belong to NF_δ.

$$\textsc{Field/Par}\ \frac{}{E\,|\,\mathcal{N} \vdash (P.x : T) \in \mathrm{NF}_\delta} \qquad \textsc{Field/Def}\ \frac{E\,|\,\mathcal{N} \vdash t \in \mathrm{NF}_\delta}{E\,|\,\mathcal{N} \vdash (P.x : T := t) \in \mathrm{NF}_\delta} \qquad \textsc{Mod}\ \frac{\forall i \in [1 \ldots n],\ E\,|\,\mathcal{N} \vdash e_i \in \mathrm{NF}_\delta}{E\,|\,\mathcal{N} \vdash P : \langle e_1 \ldots e_n \,|\, \mathcal{N}\rangle \in \mathrm{NF}_\delta}$$

$$\textsc{ModFun}\ \frac{E, (v : S)\,|\,\mathcal{N} \vdash (P\,v) : S' \in \mathrm{NF}_\delta}{E\,|\,\mathcal{N} \vdash P : (v : S) \Rightarrow S' \in \mathrm{NF}_\delta}$$

Weakly-normalizing terms

We define $t\downarrow$ (t is weakly δ-normalizing) inductively by :

$$\frac{E\,|\,\mathcal{N} \vdash t \in \mathrm{NF}_\delta}{E\,|\,\mathcal{N} \vdash t\downarrow} \qquad \frac{E\,|\,\mathcal{N} \vdash t \rhd_\delta t' \qquad E\,|\,\mathcal{N} \vdash t'\downarrow}{E\,|\,\mathcal{N} \vdash t\downarrow}$$

We extend $\downarrow$ on structures and fields. Term parameters always normalize, term definitions normalize if and only if their body normalizes. Finally, modules normalize if and only if all fields of the module normalize.

$$\textsc{Menv}\ \frac{E, (P : S[:= S'])\,|\,\mathcal{N} \cup \mathcal{N}' \vdash \mathrm{ok} \qquad E\,|\,\mathcal{N} :: P \vdash S\downarrow}{E, (P : S[:= S'])\,|\,\mathcal{N} \cup \mathcal{N}' \vdash P : S\downarrow}$$

$$\textsc{Mdot}\ \frac{E\,|\,\mathcal{N} \vdash P : \langle e_1 \ldots e_n \,|\, \mathcal{N}_P\rangle\downarrow \qquad (P.P_i : S) \in \langle e_1 \ldots e_n\rangle}{E\,|\,\mathcal{N} \vdash P.P_i : S\downarrow}$$

$$\textsc{MApp}\ \frac{E\,|\,\mathcal{N} \vdash P_1 : (v : S_1) \Rightarrow S\downarrow \qquad E\,|\,\mathcal{N} \vdash P_2 : S_1'\downarrow \qquad E\,|\,\mathcal{N} :: \# \vdash \{v/P_2\}S^\#\downarrow}{E\,|\,\mathcal{N} \vdash (P_1\,P_2) : \{v/P_2\}S\downarrow}$$

$$\textsc{Struct}\ \frac{\forall i \in [1 \ldots n] \qquad E, e_1, \ldots, e_{i-1}\,|\,\mathcal{N} \cup \mathcal{N}'_{i-1} \vdash e_i\downarrow}{E\,|\,\mathcal{N} \vdash \langle e_1 \ldots e_n \,|\, \mathcal{N}'\rangle\downarrow}$$

$$\textsc{StructFun}\ \frac{E, (v : S) \,|\, \mathcal{N} \vdash S' \downarrow}{E \,|\, \mathcal{N} \vdash (v : S) \Rightarrow S' \downarrow} \qquad \textsc{Field/Par}\ \frac{}{E \,|\, \mathcal{N} \vdash (P.x : T) \downarrow} \qquad \textsc{Field/Def}\ \frac{E \,|\, \mathcal{N} \vdash t \downarrow}{E \,|\, \mathcal{N} \vdash (P.x : T := t) \downarrow}$$

$$\textsc{Field/Mod}\ \frac{E \,|\, \mathcal{N} :: P \vdash S \downarrow}{E \,|\, \mathcal{N} \vdash (P : S[:= S']) \downarrow}$$

Lemma 3.4.1 *For all environment $E \,|\, \mathcal{N}$, terms t and T, and qualified identifiers P.x and P'.x', such that we have:*

- $E \,|\, \mathcal{N} \vdash P.x : T$, $\mathcal{N}(P.x) = P'.x'$, $(P'.x' : t := T) \in E$
- $E \,|\, \mathcal{N} \vdash t \downarrow$

then we have $E \,|\, \mathcal{N} \vdash P.x \downarrow$

Proof. By rule Delta/Def, we have $E \,|\, \mathcal{N} \vdash P.x \rhd_\delta t$, and hence we conclude on $E \,|\, \mathcal{N} \vdash P.x \downarrow$ by construction.

□

Lemma 3.4.2 *For all environment $E \,|\, \mathcal{N}$, terms t and T, such that $E \,|\, \mathcal{N} \vdash t : T$ and for all $P_i.x_i \in FQI(t)$, $E \,|\, \mathcal{N} \vdash P_i.x_i \downarrow$, we have $E \,|\, \mathcal{N} \vdash t \downarrow$.*

Proof. By induction on the structure of the term t. If $t \equiv s$ for some $s \in S$ or $t \equiv v$ then it is trivial. If $t \equiv P_i.x_i$ then we have $E \,|\, \mathcal{N} \vdash t \downarrow$ by hypothesis. If $t \equiv (u_1\, u_2)$ then we have by induction hypothesis that $E \,|\, \mathcal{N} \vdash u_1 \downarrow$ and $E \,|\, \mathcal{N} \vdash u_1 \downarrow$. Let u' (resp. u'') be the δ normal form u_1 (resp. u_2), then we have $E \,|\, \mathcal{N} \vdash (u'\, u'') \downarrow$ and hence $E \,|\, \mathcal{N} \vdash t \downarrow$. The other cases are dealt similarly.

□

Lemma 3.4.3 *If $E \,|\, \mathcal{N} \vdash P : \langle e_1 \dots e_n \,|\, \mathcal{N}_P \rangle \downarrow$, then $\forall P.P'.x \in QI(\langle e_1 \dots e_n \,|\, \mathcal{N}_P \rangle)$ $E \,|\, \mathcal{N} \vdash P.P'.x \downarrow$*

Proof. By construction, $P.P'.x$ points either to an axiom and then $E \,|\, \mathcal{N} \vdash P.P'.x \downarrow$, or to a definition which body normalizes and then $E \,|\, \mathcal{N} \vdash P.P'.x \downarrow$. If $P.P'.x$ is an indirection that belongs to the domain of $\mathcal{N}_P$ then we get the result by the rule Delta/FieldDef or Delta/FieldPar.

□

In the following, we use an alternative characterization of syntactic class of paths:

$$P := Top \mid v \mid P.p \mid P\overrightarrow{P_i}$$

Hence, without loss of generality a path can always be written as $P(\overrightarrow{P_i}).P'$, where P is of the form $Top.p_1 \ldots p_n$ or $v.p_1 \ldots p_n$, $\overrightarrow{P_i}$ is a vector of general paths and P' is a general path. This notation highlights the outermost path application. We also need to be able to discriminate paths with respect to path applications, to do it we define the notion of pure and impure path.

Definition 3.4.2 *(Impure and pure paths)*
Let P be a path, we say that P is impure if and only if $P \equiv P'(\overrightarrow{P_i}).P''$ where $\overrightarrow{P_i}$ is not the empty vector. We call pure path a path P such that $P \equiv Top.p_1 \ldots p_n$ or $P \equiv v.p_1 \ldots p_n$.

Definition 3.4.3 *(Impure and pure qualified identifier)*
Let $P.x$ be a qualified identifier, we say that $P.x$ is impure (resp. pure) if and only if P is impure (resp. pure).

Now we show that normalizability is closed under path application and module path substitution.

Lemma 3.4.4 *If $E \mid \mathcal{N} \vdash P : (v : S_1) \Rightarrow S_2 \downarrow$ and $E \mid \mathcal{N} \vdash P' : S' \downarrow$, then $E \mid \mathcal{N} \vdash PP' : \{v/P'\}S_2 \downarrow$*

Proof. By induction on the size of the functor P, $\mu((v : S_1) \Rightarrow S_2)$. We need to prove that the hypothesis of the lemma implies $E \mid \mathcal{N} :: \# \vdash \{v/P'\}S_2^{\#} \downarrow$. Hence, it is sufficient to prove that every term subfield of the latter structure normalizes. More formally, we need that $E' \mid \mathcal{N}' \vdash \{v/P'\}e \downarrow$ where e is the considered term field and $E' \mid \mathcal{N}'$ is the environment built from the decomposition of the structure $\{v/P'\}S_2^{\#}$.

If the considered field is a term parameter, then we get the result by construction. In case of a term definition with term t as body, we know that $t \downarrow$ and we want to prove that $\{v/P'\}t \downarrow$. Let t' be the normal form of t, we use the Lemma 3.4.2 to reason on the form of the qualified identifiers $P''.x \in FQI(t')$. Note that before the substitution these qualified identifiers point to parameters since t' is the normal form of t. Without loss of generality, we have $P''.x \equiv P_1.P_2 \ldots P_n.x'$ where each P_i is either qualifiers separated by dots, or a path application. Considering that each intermediate path $P_1 \ldots P_i$ has type S_i, we prove by induction on n, for all i $E' \mid \mathcal{N}' \vdash \{v/P'\}P_1 \ldots P_i : \{v/P'\}S_i \downarrow$.
For $i = 1$, we consider the following cases for P_1

- (var) $P_1 \equiv v$, we have by hypothesis that $E \mid \mathcal{N} \vdash P' : S' \downarrow$.
- (app) $P_1 \equiv v(\overrightarrow{M})$ where $\overrightarrow{M}$ are general module paths.
  In order to prove that $E' \mid \mathcal{N}' \vdash \{v/P'\}v(\overrightarrow{M}) : \{v/P'\}S_1 \downarrow$, we use an additional induction on the length of $\overrightarrow{M}$.
  If this length is null then we have $P' : S' \downarrow$ by hypothesis.

Otherwise, let $\overrightarrow{M} \equiv \overrightarrow{M'}M''$, we have $E' \,|\, \mathcal{N}' \vdash \{v/P'\}v(\overrightarrow{M'}) : (v'' : S'') \Rightarrow \{v/P'\}S_1 \downarrow$.
Suppose that $E' \,|\, \mathcal{N}' \vdash M'' : S_{M''}$, we have that $E' \,|\, \mathcal{N}' \vdash M'' : S_{M''} \downarrow$ by constructor MAPP.
Now, if M'' contains at least a path application, then we have by main induction hypothesis $E' \,|\, \mathcal{N}' \vdash \{v/P'\}M'' : \{v/P'\}S_{M''} \downarrow$ since $\mu(S_{M''}) < \mu((v : S_1) \Rightarrow S_2)$. In the other case M'' is either a variable or a pure path, and then we have $E' \,|\, \mathcal{N}' \vdash \{v/P'\}M'' : \{v/P'\}S_{M''} \downarrow$ by construction. Having a normalizing functor $\{v/P'\}v(\overrightarrow{M'})$ and a normalizing module $\{v/P'\}M''$, we use the main induction hypothesis to get the desired conclusion.

We suppose the property to be true for $i = k$, and we prove it for $i = k + 1$. We consider the following cases for P_{k+1}:

- (dot) $P_{k+1} \equiv p_1 \dots p_n$, by constructor MDOT.
- (app) $P_{k+1} \equiv P''(\overrightarrow{M})$, by induction hypothesis and constructor MDOT we have:

$$E' \,|\, \mathcal{N}' \vdash P_1 \dots P_k.P'' : \overrightarrow{(v'' : S'')} \Rightarrow S'_{k+1} \downarrow$$

We do the proof by induction on the number of arguments as in the previous case, and we conclude by main induction hypothesis since the considered functor is smaller than the functor P.

□

Lemma 3.4.5 *For all environment $E \,|\, \mathcal{N}$ and terms t and T such that $E \,|\, \mathcal{N} \vdash t : T$ we have $E \,|\, \mathcal{N} \vdash t \downarrow$.*

Proof. By induction on the structure of the term t. If t is an impure qualified identifier then we use Lemmas 3.4.3 and 3.4.4, and for all other cases we get the result either by definition or by induction hypothesis.

□

Algorithmic content of the proof of weak-normalization:

Now, we extract, from the proof of weak-normalization, a function nf_δ that takes an environment and a term and returns the term in δ normal form. Without loss of generality, we consider that all fields of the environment are in δ normal form. In the following program, we use the notation $e \underline{\in}_{\mathcal{N}} E$ to denote the membership modulo indirections of the field e to the environment E

This first part comes from the induction on the term structure in Lemma 3.4.5:

$$\begin{array}{lcl}
\mathrm{nf}_\delta(E\,|\,\mathcal{N}, v) & = & v \\
\mathrm{nf}_\delta(E\,|\,\mathcal{N}, s) & = & s \\
\mathrm{nf}_\delta(E\,|\,\mathcal{N}, P.x) & = & \text{if } (P.x : T) \in_{\mathcal{N}} E \text{ then } \mathcal{N}(P.x) \\
\mathrm{nf}_\delta(E\,|\,\mathcal{N}, P.x) & = & \text{if } (P.x : T := t) \in_{\mathcal{N}} E \text{ then } t \\
\mathrm{nf}_\delta(E\,|\,\mathcal{N}, \lambda v : T.t) & = & \lambda v :\, \mathrm{nf}_\delta(E\,|\,\mathcal{N}, T).\mathrm{nf}_\delta(E, v : T\,|\,\mathcal{N}, t) \\
\mathrm{nf}_\delta(E\,|\,\mathcal{N}, \forall v : T.U) & = & \forall v :\, \mathrm{nf}_\delta(E\,|\,\mathcal{N}, T).\mathrm{nf}_\delta(E, v : T\,|\,\mathcal{N}, U) \\
\mathrm{nf}_\delta(E\,|\,\mathcal{N}, (t\,u)) & = & (\mathrm{nf}_\delta(E\,|\,\mathcal{N}, t)\,\mathrm{nf}_\delta(E, \,|\,\mathcal{N}, u))
\end{array}$$

For module access (Lemma 3.4.3), we consider the two cases pure or impure path:

$$\begin{array}{lcl}
\mathrm{nf}_\delta(E\,|\,\mathcal{N}, P.P'.x) & = & \text{if } E\,|\,\mathcal{N} \vdash P : \langle e_1 \dots e_n\,|\,\mathcal{N}_p\rangle \text{ and } P \text{ is pure then} \\
& & \quad \text{if } (P.P'.x : T) \in_{\mathcal{N}_p} \langle e_1 \dots e_n\rangle \text{ then } \mathcal{N}_p(P.P'.x) \\
& & \quad\quad \text{if } (P.P'.x : T := t) \in_{\mathcal{N}_p} \langle e_1 \dots e_n\rangle \text{ then } t
\end{array}$$

$$\begin{array}{lcl}
\mathrm{nf}_\delta(E\,|\,\mathcal{N}, P.P'.x) & = & \text{if } E\,|\,\mathcal{N} \vdash P : \langle e_1 \dots e_n\,|\,\mathcal{N}_p\rangle \text{ and } P \equiv P_0(\overrightarrow{P_i}).P_k \text{ is impure then} \\
& & \quad \text{let } P : \langle e'_1 \dots e'_n\,|\,\mathcal{N}_p\rangle = \mathrm{nf}_\delta(E\,|\,\mathcal{N}, P : \langle e_1 \dots e_n\,|\,\mathcal{N}_p\rangle) \\
& & \quad\quad \text{if } (P.P'.x : T) \in_{\mathcal{N}_p} \langle e'_1 \dots e'_n\rangle \text{ then } \mathcal{N}_p(P.P'.x) \\
& & \quad\quad\quad \text{if } (P.P'.x : T := t) \in_{\mathcal{N}_p} \langle e'_1 \dots e'_n\rangle \text{ then } t
\end{array}$$

Path applications normalization (Lemma 3.4.4) is decomposed in two cases dot and app:

$$\begin{array}{lcl}
\mathrm{nf}_\delta(E\,|\,\mathcal{N}, P_0(\overrightarrow{P_i}).P_k : S) & = & \text{if } E\,|\,\mathcal{N} \vdash P_0(\overrightarrow{P_i}) : S' \text{ then} \\
& & \quad \mathrm{dot}(\mathrm{nf}_\delta(E\,|\,\mathcal{N}, P_0(\overrightarrow{P_i}) : S'), P_k) \\
\mathrm{nf}_\delta(E\,|\,\mathcal{N}, P_0(\overrightarrow{P_i}) : S) & = & \text{if } E\,|\,\mathcal{N} \vdash P_0 : (v_i : S_i) \Rightarrow S' \text{ and } \overrightarrow{E\,|\,\mathcal{N} \vdash P_i : S'_i} \text{ then} \\
& & \quad P_0(\overrightarrow{P_i}) : \mathrm{subst}(E\,|\,\mathcal{N}, v_n, \dots \mathrm{subst}(E\,|\,\mathcal{N}, v_1, \mathrm{nf}_\delta(E\,|\,\mathcal{N}, P_1 : S'_1), S') \dots)
\end{array}$$

The auxiliary function subst takes as argument an environment, a path variable, a normalized module, and the object to substitute that can be either a structure, a field, or a term. This function returns the δ normal form of the substituted object. we denote by σ the substitution

$\{v/P\}$:

On structures:

$$\begin{array}{l}\mathrm{subst}(E\,|\,\mathcal{N}, v, P : S, \langle e_1 \ldots e_n \,|\, \mathcal{N}'\rangle) \quad = \\ \qquad \langle \mathrm{subst}(E\,|\,\mathcal{N}, v, P : S, e_1) \ldots \mathrm{subst}(E\,|\,\mathcal{N}, v, P : S, e_n) \,|\, \sigma\mathcal{N}'\rangle \\ \mathrm{subst}(E\,|\,\mathcal{N}, v, P : S, (v_1 : S_1) \Rightarrow S_2) \quad = \\ \qquad (v_1 : \mathrm{subst}(E\,|\,\mathcal{N}, v, P : S, S_1)) \Rightarrow \mathrm{subst}(E\,|\,\mathcal{N}, v, P : S, S_2)\end{array}$$

On fields:

$$\begin{array}{lcl}\mathrm{subst}(E\,|\,\mathcal{N}, v, P : S, (P' : S_1 := S_2)) & = & \\ \qquad (\sigma P' : \mathrm{subst}(E\,|\,\mathcal{N}, v, P : S, S_1) := \mathrm{subst}(E\,|\,\mathcal{N}, v, P : S, S_2)) & & \\ \mathrm{subst}(E\,|\,\mathcal{N}, v, P : S, (P' : S_1)) & = & (\sigma P' : \mathrm{subst}(E\,|\,\mathcal{N}, v, P : S, S_1)) \\ \mathrm{subst}(E\,|\,\mathcal{N}, v, P : S, (P'.x : T := t)) & = & \\ \qquad (\sigma P'.x : \mathrm{subst}(E\,|\,\mathcal{N}, v, P : S, T) := \mathrm{subst}(E\,|\,\mathcal{N}, v, P : S, t)) & & \\ \mathrm{subst}(E\,|\,\mathcal{N}, v, P : S, (P'.x : T)) & = & (\sigma P'.x : \mathrm{subst}(E\,|\,\mathcal{N}, v, P : S, T))\end{array}$$

On terms:

$$\mathrm{subst}(E\,|\,\mathcal{N}, v, P : S, t) \quad = \quad \mathrm{nf}_\delta(E\,|\,\mathcal{N}, \sigma t)$$

The auxiliary function dot:

$$\begin{array}{l}\mathrm{dot}(E\,|\,\mathcal{N}, P : \langle e_1 \ldots e_n \,|\, \mathcal{N}_P\rangle, P') \quad = \\ \qquad \text{If } P' \text{ is pure and } (P.P' : S) \in \langle e_1 \ldots e_n\rangle \text{ then } (P.P' : S) \\ \\ \mathrm{dot}(E\,|\,\mathcal{N}, P : \langle e_1 \ldots e_n \,|\, \mathcal{N}_P\rangle, P_0(\overrightarrow{P_i}).P') \quad = \\ \qquad \text{if } E\,|\,\mathcal{N} \vdash P.P_0 : (v_i : S_i) \Rightarrow S' \text{ and } \overrightarrow{E\,|\,\mathcal{N} \vdash P_i : S_i'} \text{ then} \\ \mathrm{dot}(E\,|\,\mathcal{N}, P.P_0(\overrightarrow{P_i}) : \mathrm{subst}(E\,|\,\mathcal{N}, v_n, \ldots \mathrm{subst}(E\,|\,\mathcal{N}, v_1, \mathrm{nf}_\delta(E\,|\,\mathcal{N}, P_1 : S_1'), S') \ldots), P')\end{array}$$

3.4.3 Translation from $\mathcal{HM}$ to $\mathcal{M}$

Removing Sealing: the system $\mathcal{HM}_{/S}$

As in Section 3.2, we modify the rule Env/ModDef so that a transparent signature is given to the module definition:

$$\textsc{Env/ModDef}\quad \frac{E\,|\,\mathcal{N} :: P \vdash S : \mathrm{struct}}{E, (P : S := S)\,|\,\mathcal{N} \vdash \mathrm{ok}}$$

The system $\mathcal{HM}$ with this restricted Env/ModDef rule is denoted $\mathcal{HM}_{/S}$. As we have done for the systems $\mathcal{M}$ and $\mathcal{M}_{/\mathcal{S}}$, we can prove that both systems are equivalent in term of expressiveness, since we can always postpone the subtyping check done in premise of the rule Env/ModDef at access time.

Removing functors:

In this section, we prove that the system $\mathcal{HM}_{/S}$ is conservative with respect to the system $\mathcal{M}$. Without loss of generality, we consider environments of the system $\mathcal{HM}_{/S}$ where there is no "interactive functors in construction". For instance, we consider the following environment where the functor is declared:

$$(Top.F : (v : S) \Rightarrow \langle (Top.F\, v).x : T := t \ldots \rangle) \,|\, Top$$

but not this one:

$$(v : S), (Top.F\, v).x : T := t, \ldots \,|\, Top :: (Top.F\, v)$$

This restriction allows us to avoid some tedious namespace renaming during the translation.

Definition 3.4.4 *Let $E \,|\, \mathcal{N}$ be an environment, we say that the environment is functorialy closed if and only if the environment of declarations E does not contain any module variable declarations and the namespace $\mathcal{N}$ does not contain path variable declarations.*

As we have done for previous systems, we define a binary relation between functorialy closed environments of the system $\mathcal{HM}_{/S}$ and environments of the system $\mathcal{M}$. Basically, a $\mathcal{HM}_{/S}$ environment is in relation with its translated $\mathcal{M}$ environment. The translation removes functor declaration, keeps from applied functors a renamed form of the axioms they declare, and accumulates a substitution to propagate this renaming. To do it, we compute the substituted δ normal form of each terms. If the result is an impure qualified identifier then it means that the latter points to an axiom declared within a functor. Hence, we give it a fresh pure name, we add it in the $\mathcal{M}$ environment, and we update the substitution. To illustrate this translation take the following development:

```
Module F (v : S).
  Parameter a : Type.
  Definition b := (fun x : a) => v.t.
End F.

Module M.
  Definition t := Type.
End M.

Definition a1 := (F M).a.
Definition b1 := (F M).b.
```

We suppose that S is the abbreviation for the structure containing the field Parameter *t : Type.* We build in parallel the $\mathcal{HM}_{/S}$ environment and the $\mathcal{M}$ environment:

- we start from the empty environments and empty substitution.
- we add the functor F in the $\mathcal{HM}_{/S}$ environment, the $\mathcal{M}$ environment stays empty.
- we add the module M in the $\mathcal{HM}_{/S}$ environment, and we add its δ normal form in the $\mathcal{M}$ environment.
- we compute the substituted δ normal form of *(F M).a*. It gives the impure qualified identifier *(F M).a*, we add in the $\mathcal{M}$ environment the parameter *(Top.a : Type)* and we extend the substitution with $\{(FM).a/Top.a\}$. Now, we add the substituted δ normal form of the definition $Top.a1$ in the $\mathcal{M}$ environment and its original form in the $\mathcal{HM}_{/S}$ environment.
- we compute the substituted δ normal form of *(F M).b*. It gives *(fun x : Top.a) => Type* which do not contain any impure qualified identifier, hence we can add the corresponding definition in $\mathcal{M}$ environment and its original form in the $\mathcal{HM}_{/S}$ environment.

Definition 3.4.5 *Let ρ be a substitution that maps impure qualified identifiers to pure qualified identifiers, we define inductively the binary relation $\underset{\rho}{\leq}$ between environments of the system $\mathcal{M}$ and functorialy closed environments of the system $\mathcal{HM}_{/S}$, with the following constructors:*

$$\textsc{Empty}\quad \frac{}{(\epsilon \,|\, Top)_{\mathcal{M}} \quad \underset{\epsilon}{\leq} \quad (\epsilon \,|\, Top)_{\mathcal{HM}_{/S}}}$$

$$\textsc{Rename}\quad \frac{E \,|\, \mathcal{N} \quad \underset{\rho}{\leq} \quad E' \,|\, \mathcal{N}' \qquad E' \,|\, \mathcal{N}' \vdash_{\mathcal{HM}_{/S}} P'.x' : T' \qquad P''.x'' = \rho(\mathrm{nf}_\delta(P'.x')) \qquad P''.x'' \text{ impure} \qquad T = \rho(\mathrm{nf}_\delta(E' \,|\, \mathcal{N}', T')) \qquad FQI(T) \text{ are pure}}{E, (P.x : T) \,|\, \mathcal{N} \quad \underset{\rho, \{P''.x''/P.x\}}{\leq} \quad E' \,|\, \mathcal{N}'}$$

The empty environments are in relation. The RENAME constructor is the most important one. It adds in the environment of the system $\mathcal{M}$ a parameter that replaces a parameter defined within a functor. If we have an environments $E \,|\, \mathcal{N}$ of the system $\mathcal{M}$ in relation with the environment $E' \,|\, \mathcal{N}'$ of the system $\mathcal{HM}_{/S}$ such that we have:

- a derivation of $P'.x' : T'$ in $\mathcal{HM}_{/S}$
- the substituted δ normal form of $P'.x'$ is an impure qualified identifier, this condition implies that $P''.x''$ is a parameter defined within a functor, and that this parameter has not been renamed yet.
- a term T which is the substituted δ normal form of T', and that only contains pure qualified identifier

then we have that the environment of $\mathcal{M}$ extended with the parameter $(P.x : T)$, for some path P chosen within $\mathcal{N}$ and x a fresh identifier, is in relation with the former environment of $\mathcal{HM}_{/S}$.

PAR

$$\frac{E \,|\, \mathcal{N} \underset{\rho}{\leq} E' \,|\, \mathcal{N}' \qquad E' \,|\, \mathcal{N}' \vdash_{\mathcal{HM}_{/S}} T' : s \qquad T = \rho(\mathrm{nf}_\delta(T')) \qquad FQI(T) \text{ are pure}}{E, (P.x : T) \,|\, \mathcal{N} \underset{\rho}{\leq} E', (P.x : T') \,|\, \mathcal{N}'}$$

DEF

$$\frac{E \,|\, \mathcal{N} \underset{\rho}{\leq} E' \,|\, \mathcal{N}' \qquad E' \,|\, \mathcal{N}' \vdash_{\mathcal{HM}_{/S}} t' : T' \qquad T = \rho(\mathrm{nf}_\delta(T')) \qquad t = \rho(\mathrm{nf}_\delta(t)) \qquad FQI(T) \text{ and } FQI(t) \text{ are pure}}{E, (P.x : T := t) \,|\, \mathcal{N} \underset{\rho}{\leq} E', (P.x : T' := t') \,|\, \mathcal{N}'}$$

For term parameter and definition, the environments are in relation if the $\mathcal{M}$ environment contains the substituted δ normal form of the declaration in the $\mathcal{HM}_{/S}$ environment.

NAMESPACE

$$\frac{E \,|\, \mathcal{N} \underset{\rho}{\leq} E' \,|\, \mathcal{N}'}{E \,|\, \mathcal{N} :: P \underset{\rho}{\leq} E' \,|\, \mathcal{N}' :: P}$$

INDIRECT

$$\frac{E \,|\, \mathcal{N} \underset{\rho}{\leq} E' \,|\, \mathcal{N}'}{E \,|\, \mathcal{N} \underset{\rho}{\leq} E' \,|\, \mathcal{N}' + (P.x \leftarrow P'.x')}$$

Indirections are not considered in the environment of $\mathcal{M}$ since they are always eliminated in the

δ normal form of a term.

$$\textsc{ModPFun}\quad \frac{E\,|\,\mathcal{N} \underset{\rho}{\leq} E'\,|\,\mathcal{N}' \qquad E'\,|\,\mathcal{N}' :: P \vdash_{\mathcal{HM}/S} S : \text{struct} \qquad S \text{ higher-order structure}}{E\,|\,\mathcal{N} \underset{\rho}{\leq} E', (P : S)\,|\,\mathcal{N}'}$$

$$\textsc{ModDFun}\quad \frac{E\,|\,\mathcal{N} \underset{\rho}{\leq} E'\,|\,\mathcal{N}' \qquad E'\,|\,\mathcal{N}' :: P \vdash_{\mathcal{HM}/S} S : \text{struct} \qquad S \text{ higher-order structure}}{E\,|\,\mathcal{N} \underset{\rho}{\leq} E', (P : S := S)\,|\,\mathcal{N}'}$$

$$\textsc{ModP}\quad \frac{E, e_1, \ldots, e_n\,|\,\mathcal{N} :: P :: \mathcal{N}_p \underset{\rho}{\leq} E', e'_1, \ldots, e'_m\,|\,\mathcal{N}' :: P :: \mathcal{N}'_P}{E, (P : \langle e_1 \ldots e_n\,|\,\mathcal{N}_p\rangle)\,|\,\mathcal{N} \underset{\rho}{\leq} E', (P : \langle e'_1 \ldots e'_m\,|\,\mathcal{N}'_p\rangle)\,|\,\mathcal{N}'}$$

$$\textsc{ModD}\quad \frac{E, e_1, \ldots, e_n\,|\,\mathcal{N} :: P :: \mathcal{N}_p \underset{\rho}{\leq} E', e'_1, \ldots, e'_m\,|\,\mathcal{N}' :: P :: \mathcal{N}'_P}{E, (P : \langle e_1 \ldots e_n\,|\,\mathcal{N}_p\rangle := \langle e_1 \ldots e_n\,|\,\mathcal{N}_p\rangle)\,|\,\mathcal{N} \underset{\rho}{\leq} E', (P : \langle e'_1 \ldots e'_m\,|\,\mathcal{N}'_p\rangle := \langle e'_1 \ldots e'_m\,|\,\mathcal{N}'_p\rangle)\,|\,\mathcal{N}'}$$

Finally, for module parameters and definitions, functors are not added in the $\mathcal{M}$ environment while first-order modules are. The difference in term of structure length, in the constructors ModP and ModD, comes from the possible presence of sub-functor fields and renamed parameters.

Lemma 3.4.6 *For all environments $E'\,|\,\mathcal{N}'$ and $E\,|\,\mathcal{N}$, if there exists ρ such that $E\,|\,\mathcal{N} \underset{\rho}{\leq} E'\,|\,\mathcal{N}'$, then for all qualified identifier $P.x$ and term T' such that $(P.x : T') \in E'$, we have $(P.x : \rho(nf_\delta(T)) \in E$*

Proof. By construction of the relation $\underset{\rho}{\leq}$.

□

Lemma 3.4.7 *For all environments $E' \,|\, \mathcal{N}'$ and $E \,|\, \mathcal{N}$, if we have :*

- $E' \,|\, \mathcal{N}' \vdash_{\mathcal{HM}_{/S}} ok$ *and* $E \,|\, \mathcal{N} \vdash_{\mathcal{M}} ok$
- $E \,|\, \mathcal{N} \underset{\rho}{\leq} E' \,|\, \mathcal{N}'$ *for some* ρ
- $E' \,|\, \mathcal{N}' \vdash_{\mathcal{HM}_{/S}} P : \langle e'_1 \dots e'_m \,|\, \mathcal{N}'_p \rangle$ *with* P *a pure path*

then we have: (1) $E \,|\, \mathcal{N} \vdash_{\mathcal{M}} P : \langle e_1 \dots e_n \,|\, \mathcal{N}_p \rangle$ *for some structure* $\langle e_1 \dots e_n \,|\, \mathcal{N}_p \rangle$*, and (2) for all qualified identifier* $P.P'.x$ *and term* T' *such that* $(P.P'.x : T') \in \langle e'_1 \dots e'_m \rangle$*, we have* $(P.P'.x : \rho(nf_\delta(T)) \in \langle e_1 \dots e_n \rangle$

Proof. By construction of the relation $\underset{\rho}{\leq}$, we see that the property (1) is induced by the constructors ModP and ModD and by applying the Lemma 3.4.6 on their premise we get the property (2).

□

Lemma 3.4.8 *For all environments $E' \,|\, \mathcal{N}'$ and $E \,|\, \mathcal{N}$, terms t and T, if we have*

- $E' \,|\, \mathcal{N}' \vdash_{\mathcal{HM}_{/S}} ok$ *and* $E \,|\, \mathcal{N} \vdash_{\mathcal{M}} ok$
- *there exists* ρ *such that* $E \,|\, \mathcal{N} \underset{\rho}{\leq} E' \,|\, \mathcal{N}'$
- $E' \,|\, \mathcal{N}' \vdash_{\mathcal{HM}_{/S}} t : T$

then we have $E \,|\, \mathcal{N} \vdash_{\mathcal{M}} \rho(nf_\delta(t)) : \rho(nf_\delta(T))$

Proof. By induction on the derivation of $E' \,|\, \mathcal{N}' \vdash_{\mathcal{HM}_{/S}} \mathrm{nf}_\delta(t) : \mathrm{nf}_\delta(T)$. If the last rule of the derivation is Term/Ax then it is trivial. If the last rule of the derivation is one of the following Term/Lambda, Term/Prod, or Term/App, then we conclude by induction hypothesis.

Suppose that the last rule is Term/Acc:

$$\frac{E' \,|\, \mathcal{N}' \vdash_{\mathcal{HM}_{/S}} \mathrm{ok} \qquad (P.x : T') \in E'}{E' \,|\, \mathcal{N}' \vdash_{\mathcal{HM}_{/S}} P.x : T'}$$

with $P.x \equiv \mathrm{nf}_\delta(t)$ and $T' \equiv \mathrm{nf}_\delta(T)$. We know that $P.x$ points to an axiom in the environment E' or else it would not be in δ normal form. By Lemma 3.4.6, we have $(P.x : \rho(T')) \in E$. Hence, we conclude on $E \,|\, \mathcal{N} \vdash_{\mathcal{M}} P.x : \rho(T)$ by applying the rule Term/Acc.

Suppose that the last rule is Term/Field:

$$\frac{\textsc{Term/Field}}{E' \,|\, \mathcal{N}' \vdash_{\mathcal{HM}_{/S}} P : \langle e'_1 \dots e'_m \,|\, \mathcal{N}_P \rangle \qquad (P.P'.x : T') \in \langle e'_1 \dots e'_m \rangle}{E' \,|\, \mathcal{N}' \vdash_{\mathcal{HM}_{/S}} P.P'.x : T'}$$

with $P.P'.x \equiv \mathrm{nf}_\delta(t)$ and $T' \equiv \mathrm{nf}_\delta(T)$. We know that $P.P'.x$ points to an axiom in the structure $\langle e_1 \dots e_n \rangle$ or else it would not be in δ normal form. Here, we need to consider two cases: P is either a pure path or an impure path.

- If P is an impure path. We have by construction, that $\rho(P.P'.x) = P''.x'$ and $(P''.x' : \rho(T)) \in E$. We apply the rule Term/Acc on $E \,|\, \mathcal{N} \vdash_{\mathcal{M}} \mathrm{ok}$ and we get $E \,|\, \mathcal{N} \vdash_{\mathcal{M}} P''.x' : T'$.
- If P is a pure path. We have by Lemma 3.4.7 that $E \,|\, \mathcal{N} \vdash_{\mathcal{HM}_{/S}} P : \langle e_1 \dots e_n \,|\, \mathcal{N}_P \rangle$ and $(P.P'.x : \rho(T')) \in \langle e_1 \dots e_n \,|\, \mathcal{N}_P \rangle)$. We use the rule Term/Field and we conclude on $E \,|\, \mathcal{N} \vdash_{\mathcal{M}} P.P'.x : \rho(T')$.

□

Lemma 3.4.9 *For all environment $E' \,|\, \mathcal{N}'$, such that $E' \,|\, \mathcal{N}' \vdash_{\mathcal{HM}_{/S}} ok$, there exist ρ and $E \,|\, \mathcal{N}$ such that $E \,|\, \mathcal{N} \underset{\rho}{\leq} E' \,|\, \mathcal{N}'$ and $E \,|\, \mathcal{N} \vdash_{\mathcal{M}} ok$;*

Proof. By induction on the derivation of $E' \,|\, \mathcal{N}' \vdash_{\mathcal{HM}_{/S}} \mathrm{ok}$.

- If the environment is empty then it is trivial.
- If the last rule of the derivation is either Env/Namespace or Env/Indirect then we conclude by induction hypothesis.
- If the last rule of the derivation is either Env/Def or Env/Par. We do the proof for Env/Par, the case Env/Def is dealt similarly.

$$\frac{E' \,|\, \mathcal{N}' \vdash_{\mathcal{HM}_{/S}} \mathrm{ok} \qquad E' \,|\, \mathcal{N}' \vdash_{\mathcal{HM}_{/S}} T : s \qquad s \in S \qquad P \in \mathcal{N}}{E', (P.x : T) \,|\, \mathcal{N}' \vdash_{\mathcal{HM}_{/S}} \mathrm{ok}}$$

  In order to be able apply the Lemma 3.4.8 on $E' \,|\, \mathcal{N}' \vdash_{\mathcal{HM}_{/S}} T : s$, we need to enrich the ρ and $E \,|\, \mathcal{N}$, obtained by induction hypothesis on the derivation of $E' \,|\, \mathcal{N}' \vdash_{\mathcal{HM}_{/S}} \mathrm{ok}$, with the renamed parameters that correspond to the impure qualified identifiers that appears freely in $\mathrm{nf}_\delta(E' \,|\, \mathcal{N}', T)$.
  Let T' be the δ normal form of T, and let Þ be the set of impure qualified identifiers that appear freely in T'. We prove by induction on the on the cardinal of Þ that there exists E_{par} and ρ_{par} such that $E, E_{par} \,|\, \mathcal{N} \underset{\rho\rho_{par}}{\leq} E' \,|\, \mathcal{N}'$ and $\rho\rho_{par}(T')$ contains only pure qualified identifier. If the cardinal of Þ is 0 then we take $E_{par} = \epsilon$ and ρ_{par} the empty substitution. We suppose the properties to be true for Þ of cardinal n and we do the proof for $n+1$. Suppose that $P_{n+1}.x_{n+1} \in$ Þ and $E' \,|\, \mathcal{N}' \vdash_{\mathcal{HM}_{/S}} P_{n+1}.x_{n+1} : T_{n+1}$ where T_{n+1} is in δ normal form. We have two easy cases:

– if $\rho\rho_{par}(P_{n+1}.x_{n+1})$ is a pure qualified identifier then, we have already renamed and added the corresponding parameter in the $\mathcal{M}$ environment. Hence, we conclude by induction hypothesis.

– if $\rho\rho_{par}(T_{n+1})$ contains only pure qualified identifiers then we conclude on:

$$E, E_{par}, (P.x' : \rho\rho_{par}(T_{n+1})) \,|\, \mathcal{N} \underset{\rho\rho_{par}\{P_{n+1}.x_{n+1}/P.x'\}}{\leq} E' \,|\, \mathcal{N}'$$

by constructor RENAME[6]. We have that $\rho\rho_{par}\{P_{n+1}.x_{n+1}/P.x'\}(T')$ contains only pure qualified identifier by construction.

Now, if $\rho\rho_{par}(T_{n+1})$ contains impure qualified identifiers, then we recursively do the process above for T_{n+1} and $\mathfrak{P}$ the set of impure qualified identifiers that appear freely in T_{n+1}.We get that there exists E_{n+1} and ρ_{n+1} such that

$$E, E_{par}, E_{n+1} \,|\, \mathcal{N} \underset{\rho\rho_{par}\rho_{n+1}}{\leq} E' \,|\, \mathcal{N}'$$

and $\rho\rho_{par}\rho_{n+1}(T_{n+1})$ contains only pure qualified identifier. By constructor RENAME we can add the renamed and substituted parameter $(P.x' : \rho\rho_{par}\rho_{n+1}(T_{n+1}))$ in the $\mathcal{M}$ environment. This recursive process terminates for the same reason as the δ reduction normalizes (i.e. induction on the size of the functor and nested induction on the form of the impure path).
Now that we have built the right $\mathcal{M}$ environment $E, E_{par} \,|\, \mathcal{N}$ and substitution $\rho\rho_{par}$, we apply the Lemma 3.4.8 and we get:

$$E, E_{par} \,|\, \mathcal{N} \vdash_{\mathcal{M}} \rho\rho_{par}(T') : s$$

Finally, we apply the rule ENV/PAR to build the desired well-formed environment.

- If the last rule of the derivation is either ENV/MODDEF or ENV/MODPAR then we need to consider two cases, either we add a higher-module or we add a first-order module.
If we add a higher-module then we get the result by induction hypothesis. Suppose that the last rule is ENV/MODPAR and the module parameter is a first-order module:

$$\text{ENV/MODPAR}\quad \frac{E' \,|\, \mathcal{N}' \vdash_{\mathcal{HM}_{/S}} \text{ok} \qquad E' \,|\, \mathcal{N}' :: P \vdash_{\mathcal{HM}_{/S}} \langle e'_1 \ldots e'_m \,|\, \mathcal{N}'_P \rangle : \text{struct}}{E', (P : \langle e'_1 \ldots e'_m \,|\, \mathcal{N}'_P \rangle) \,|\, \mathcal{N}' \vdash_{\mathcal{HM}_{/S}} \text{ok}}$$

By inversion on the premise we have that $E', e'_1, \ldots, e'_m \,|\, \mathcal{N}' :: P :: \mathcal{N}'_P \vdash_{\mathcal{HM}_{/S}} \text{ok}$, hence we have by induction hypothesis that there exists ρ and $E, e_1, \ldots, e_n \,|\, \mathcal{N} :: P :: \mathcal{N}_P$ such that

$$E, e_1, \ldots, e_n \,|\, \mathcal{N} :: P :: \mathcal{N}_P \vdash_{\mathcal{M}} \text{ok} \qquad (1)$$

[6]Here we take the qualifier P that is valid in $\mathcal{N}$ by construction, and a fresh identifier x'. It is important to choose the qualifier used for the translated field (i.e. $P.x : T$), in order to respect the first-order module hierarchy.

and

$$E, e_1, \ldots, e_n \,|\, \mathcal{N} :: P :: \mathcal{N}_P \underset{\rho}{\leq} E', e'_1, \ldots, e'_m \,|\, \mathcal{N}' :: P :: \mathcal{N}'_P \qquad (2)$$

We apply the rule Struct/Env and Env/ModPar on (1) and we get:

$$E, (P : \langle e_1 \ldots e_n \,|\, \mathcal{N}_P \rangle) \,|\, \mathcal{N} \vdash_{\mathcal{M}} \text{ok}$$

We apply the constructor ModP on (2) and we get:

$$E, (P : \langle e_1 \ldots e_n \,|\, \mathcal{N}_P \rangle) \,|\, \mathcal{N} \underset{\rho}{\leq} E', (P : \langle e'_1 \ldots e'_m \,|\, \mathcal{N}'_P \rangle) \,|\, \mathcal{N}'$$

The case Env/ModDef is dealt similarly.

□

Corollary 3.4.1 *For all environment $E' \,|\, \mathcal{N}'$, terms t and T, if $E' \,|\, \mathcal{N}' \vdash_{\mathcal{HM}_{/S}} t : T$ and $nf_\delta(E' \,|\, \mathcal{N}', T)$ contains only pure qualified identifier, then there exists a substitution ρ and an environment $E \,|\, \mathcal{N}$ such that $E \,|\, \mathcal{N} \underset{\rho}{\leq} E' \,|\, \mathcal{N}'$ and $E \,|\, \mathcal{N} \vdash_{\mathcal{M}} \rho(nf_\delta(t)) : nf_\delta(T)$.*

Chapter 4

Extensions of the system

In this chapter, we give extensions of the module system that are a part of the concrete implementation. In a first time, we describe how to extend the system with structure abbreviations, that roughly correspond to signature abbreviations. Then, we give a formal description of the Δ-equivalence introduced in Section 2.3.5. This new relation allows to quotient the namespace and derive canonical names. Finally, we introduce a new inlining notion that allows, when applying a functor, the automation of δ-reduction of qualified identifier selected by the user.

4.1 Structure abbreviations

The first main extension of our system is to take in account a new kind of fields that corresponds to named structures, called structure abbreviation. Since our system is based on the unique notion of structure, the abbreviations are used in two ways. The first one is the most common, and corresponds to signature abbreviation that are reused for sealing. The second one is more specific to our approach. It allows to build step by step a concrete implementation of a given theory. In other words, we start from a structure that specifies abstractly a theory. Then using merge and application, we successively derive intermediate named structures, containing defined fields, that finally correspond to instantiations of the former theory. Now, these fully defined structures can be used to define modules, and the intermediate structures can be used either as starting points for other theory instantiations or as precise translucent signatures.

Syntax

$e + + = (P = S)$

A structure abbreviation is a part of the syntactic class of fields. It is the association of a path and a structure.

Typing rules

$$\textsc{Env/Abbrev}\quad \frac{E\,|\,\mathcal{N} :: P \vdash S : \text{struct}}{E, (P = S)\,|\,\mathcal{N} \vdash \text{ok}} \qquad \textsc{Struct/Abbrev}\quad \frac{E\,|\,\mathcal{N} :: P' \vdash \text{ok} \qquad (P = S) \in E}{E\,|\,\mathcal{N} :: P' \vdash \{P/P'\}S : \text{struct}}$$

$$\textsc{Struct/ModField}\quad \frac{E\,|\,\mathcal{N} :: P' \vdash P : \langle e_1 \dots e_n\,|\,\mathcal{N}_P\rangle \qquad (P.P_i = S) \in \langle e_1 \dots e_n\rangle}{E\,|\,\mathcal{N} :: P' \vdash \{P.P_i/P'\}S : \text{struct}}$$

$$\textsc{Struct/App}\quad \frac{E\,|\,\mathcal{N} \vdash (v : S_1) \Rightarrow S : \text{struct} \qquad E\,|\,\mathcal{N} \vdash P : S_2 \qquad E\,|\,\mathcal{N} :: \# \vdash S_{2/P_2}^{\#} \subseteq S_1^{\#}}{E\,|\,\mathcal{N} \vdash \{v/P\}S : \text{struct}}$$

$$\textsc{Sub/Abbrev/Abbrev}\quad \frac{E\,|\,\mathcal{N} :: P \vdash S \subseteq S' \qquad E\,|\,\mathcal{N} :: P \vdash S' \subseteq S}{E\,|\,\mathcal{N} \vdash (P = S) \subseteq (P = S')}$$

The Env/Abbrev rule adds a new structure abbreviation in the environment when the structure is well-formed under the abbreviation path. Both Struct/Abbrev and Struct/ModField rules allow to recover the structure corresponding to an abbreviation. Finally, the new Sub rule is added to consider subtyping derivation that involves structure abbreviation fields. Here, we do not require the two structures to be equal but only to be mutual subtype. With the help of the antisymmetry of the relation $\leq_{\beta\delta}$, this induces that all term fields of the structure are equivalent. However, the fields can be declared in a different order.

The strengthening has no effect on structure abbreviation fields, hence we extend it as follows:

$$(P' = S)_{/P} \;=\; (P' = S)$$

Concluding remark

Structure abbreviation is essential for Coq users. Beside the classical code factorization that it provides, it allows a new way to build concrete structures through "refinement steps". At first sight, one might think that the addition of structure abbreviations could lead to undecidability of module type-checking as it was proven for the translucent sum calculus [24, 38]. However, this is not the case here because this result clearly depends on the design space of the module system. In fact, it occurs when the module language and the base language are not separated [50]. In this work, we build a module system where structures and modules are second class objects and hence the base language and the module language are clearly separated . Furthermore our base language offers an oracle ($=_{\beta\delta}$ and $\leq_{\beta\delta}$) that determines if two terms are convertible or subtype. Hence, it allows to build a quite straightforward algorithm to check structure subtyping.

4.2 Δ-equivalence

We have presented in Section 2.4, the scope of this new equivalence relation on names called Δ. It allows the proof writing machinery to reason modulo sharing constraints implied by the modular structure of the development. We have introduced this notion in [53], and we have implemented it in Coq v8.3. Basically, this relation is computed thanks to the strengthening steps of the module system, and is propagated through functor applications by an extended path substitution.

Syntax:

We modify the syntax of term declarations so that their name parts hold the Δ-equivalence information:

$$e := P.x \triangleright P.x : T \mid P.x \triangleright P.x : T := t \mid P : S \mid P : S := S'$$

The rest of the syntax is unmodified. Now, a term field is named by two qualified identifiers. On the left hand side of $\triangleright$ we still have the usual principal qualified identifier, and on the right hand side we have the canonical qualified identifier.

Integration in the system $\mathcal{M}$:

We show how to compute Δ-equivalence in the first-order module system. First, we illustrate it by the following example:

```
Module P.
  Definition x : T := ...
  Definition  y : U := ...
  Lemma foo : ...
End P.

Module K:=P.

Module M.
  Include K.
  Lemma bar : ...
End M.
```

In this development, we have:

- a module P that defines three fields

- a module K that is a copy of P
- a module M that extends the module K

It is clear that $P.x$, $K.x$ and $M.x$ (respectively y and foo) are three names that stand for the same definition. Hence, in our formal syntax those modules are written as follows:

$$\begin{aligned} P \quad &: \quad \langle P.x \rhd P.x : T := \ldots, P.y \rhd P.y : T := \ldots, P.foo \rhd P.foo : \ldots \mid \epsilon \rangle \\ &\quad := \langle P.x \rhd P.x : T := \ldots, P.y \rhd P.y : T := \ldots, P.foo \rhd P.foo : \ldots \mid \epsilon \rangle \end{aligned}$$

$$\begin{aligned} K \quad &: \quad \langle K.x \rhd P.x : T := P.x, K.y \rhd P.y : T := P.y, K.foo \rhd P.foo : \ldots \mid \epsilon \rangle \\ &\quad := \langle K.x \rhd P.x : T := P.x, K.y \rhd P.y : T := P.y, K.foo \rhd P.foo : \ldots \mid \epsilon \rangle \end{aligned}$$

$$\begin{aligned} M \quad &: \quad \langle M.x \rhd P.x : T := K.x, M.y \rhd P.y : T := K.y, \ldots, M.bar \rhd M.bar : \ldots \mid \epsilon \rangle \\ &\quad := \langle M.x \rhd P.x : T := K.x, M.y \rhd P.y : T := K.y, \ldots, M.bar \rhd M.bar : \ldots \mid \epsilon \rangle \end{aligned}$$

We see that we keep the canonical qualified identifier of each fields when we invoke the rule Struct/Path:

$$\textsc{Struct/Path}\ \frac{E \mid \mathcal{N} :: P \vdash P' : S}{E \mid \mathcal{N} :: P \vdash (\{P'/P\}S)_{/P'} : \text{struct}}$$

In this rule, the path substitution $\{P'/P\}$ changes the domain of the name of each field. Hence, in order to keep the canonical part, we adapt the behavior of path substitutions towards our new naming management.

Definition 4.2.1 *Let σ be a path substitution, the new behavior of the path substitution is defined as follows:*

- *On module fields, structures, and namespaces the path substitution is defined as in the original Definition 3.2.1.*
- *On term fields:*

$$\sigma(P.x \rhd P'.x : T := t) = \sigma P.x \rhd P'.x : \sigma T := \sigma t$$

$$\sigma(P.x \rhd P'.x : T) = \sigma P.x \rhd P'.x : \sigma T$$

Canonical names propagation:

If we consider the merge operator, then we need to propagate canonical names through the merging process. Take the following module types that extend the previous example:

Module Type S.
  Parameter x : T.
  Parameter y : U.
  Parameter foo :...
End S.

Module Type S_1.
  Module A:=P
End S_1.

Module Type S_2.
  Declare Module A:S.
  Module B:=A.
End S_2.

Regarding our new Struct/Path rule, the fields x, y and foo of the module $S_1.A$ (resp. $S_2.B$) have $P.x$, $P.y$ and $P.foo$ as canonical names (resp. $S_2.A.x$, $S_2.A.y$ and $S_2.A.foo$).
Now, we define S_3 to be the merge of S_1 and S_2:

Module Type $S_3 := S_1 + S_2$.

The structure bound by the path S_3 is :

$$\langle S_3.A : \langle S_3.A.x \rhd P.x : T := P.x,\ S_3.A.y \rhd P.y : T := P.y, \ldots\rangle := \langle \ldots\rangle,$$
$$S_3.B : \langle S_3.B.x \rhd P.x : T := S_3.A.x,\ S_3.B.y \rhd P.y : T := S_3.A.y, \ldots\rangle := \langle \ldots\rangle\rangle$$

Remark that the fields of the module B have not anymore canonical identifiers prefixed by A. The equivalence between the names of $S_3.A$ and P has been propagated in the whole structure.

We define the class of Δ set and a function *collect* that extracts Δ sets from structures and fields. A Δ set is a set of pairs of qualified identifiers, each pair gives qualified identifiers that are Δ-equivalent.

$$\Delta := \epsilon \mid \Delta, (P.x \rhd P'.x)$$

- *collect* on structures:

$$collect(\langle e_1 \dots e_n \,|\, \mathcal{N} \rangle) \quad = \quad \bigcup_{i \in \{1 \dots n\}} collect(e_i)$$

$$collect(((v : S) \Rightarrow S')) \quad = \quad collect(S')$$

- *collect* on module fields and structure abbreviations:

$$collect((P : S := S')) \quad = \quad collect(S)$$

$$collect((P : S)) \quad = \quad collect(S)$$

$$collect((P = S) \quad = \quad \{\,\}$$

- *collect* on term fields:

$$collect((P.x \rhd P'.x : T := t)) \quad = \quad \{(P.x \rhd P'.x)\}$$

$$collect((P.x \rhd P'.x : T)) \quad = \quad \{(P.x \rhd P'.x)\}$$

Finally, in order to propagate the information contained in Δ sets, we define the operator $\Delta \uparrow$ on structures and fields as follows:

- On structures:

$$\Delta \uparrow \langle e_1 \dots e_n \,|\, \mathcal{N} \rangle \quad = \quad \langle \Delta \uparrow e_1 \dots \Delta \uparrow e_n \,|\, \mathcal{N} \rangle$$

$$\Delta \uparrow ((v : S) \Rightarrow S') \quad = \quad = (v : \Delta \uparrow S) \Rightarrow \Delta \uparrow S'$$

- On module fields and structure abbreviations:

$$\Delta \uparrow (P : S := S') \quad = \quad (P : \Delta \uparrow S := \Delta \uparrow S')$$

$$\Delta \uparrow (P : S) \quad = \quad (P : \Delta \uparrow S)$$

$$\Delta \uparrow (P = S) \quad = \quad (P = \Delta \uparrow S)$$

- On term fields:

$$\Delta \uparrow (P.x \rhd P'.x : T := t) \quad = \quad (P.x \rhd \Delta(P'.x) : T := t)$$

$$\Delta \uparrow (P.x \rhd P'.x : T) \quad = \quad (P.x \rhd \Delta(P'.x) : T)$$

where $\Delta(P'.x)$ is $P''.x$ (resp. $P'.x$) if $(P'.x \rhd P''.x) \in \Delta$ (resp. if not).

Now, we can modify the rules of the class MERGE so that canonical names are propagated:

MERGE/MATCH

$$\frac{\begin{array}{c} name(e_1) = name(e'_1) \qquad \mathcal{I}_{e''_1} = \mathcal{I}_{e_1} \cup \mathcal{I}_{e'_1} \qquad E, \mathcal{N} \vdash e_1 \uplus e'_1 \rightsquigarrow e''_1, \Delta \\ E, e''_1 \,|\, \mathcal{N} + \mathcal{I}_{e''_1} \vdash \langle e_2 \dots e_n \,|\, \mathcal{N}_1 \rangle \uplus \langle e'_2 \dots e'_m \,|\, \mathcal{N}_2 \rangle \rightsquigarrow \langle e''_2 \dots e''_k \,|\, \mathcal{N}_3 \rangle \end{array}}{E \,|\, \mathcal{N} \vdash \langle e_1 \dots e_n \,|\, \mathcal{N}_1 + \mathcal{I}_{e_1} \rangle \uplus \langle e'_1 \dots e'_m \,|\, \mathcal{N}_2 + \mathcal{I}_{e'_1} \rangle \rightsquigarrow \langle e''_1, \Delta \uparrow e''_2 \dots \Delta \uparrow e''_k \,|\, \mathcal{N}_3 + \mathcal{I}_{e''_1} \rangle}$$

MERGE/FIELD/RIGHT

$$\frac{E \,|\, \mathcal{N} \vdash e_2 \subseteq e_1}{E \,|\, \mathcal{N} \vdash e_1 \uplus e_2 \rightsquigarrow e_2, collect(e_2)}$$

MERGE/FIELD/LEFT

$$\frac{E \,|\, \mathcal{N} \vdash e_1 \subseteq e_2}{E \,|\, \mathcal{N} \vdash e_1 \uplus e_2 \rightsquigarrow e_1, collect(e_1)}$$

We modify the rules STRUCT/APP and MOD/APP in order to perform the same kind of canonical name propagation:

STRUCT/APP

$$\frac{\begin{array}{c} E \,|\, \mathcal{N} \vdash (v : S_1) \Rightarrow S : \text{Struct} \\ E \,|\, \mathcal{N} \vdash P : S_2 \qquad E \,|\, \mathcal{N} :: \# \vdash S_2^{\#}{}_{/P_2} \subseteq \{v/P\} S_1^{\#} \qquad \Delta = collect(P : S_2) \end{array}}{E \,|\, \mathcal{N} \vdash \Delta \uparrow (\{v/P\} S) : \text{Struct}}$$

MOD/APP

$$\frac{\begin{array}{c} E \,|\, \mathcal{N} \vdash P_1 : (v : S_1) \Rightarrow S \\ E \vdash P_2 : S_2 \qquad E \,|\, \mathcal{N} :: \# \vdash S_2^{\#}{}_{/P_2} \subseteq \{v/P_2\} S_1^{\#} \qquad \Delta = collect(P : S_2) \end{array}}{E \,|\, \mathcal{N} \vdash (P_1\, P_2) : \Delta \uparrow (\{v/P_2\} S)}$$

Concluding remarks

In this section, we have added the layer of canonical names, thus we need to determine the scope of use of each layer (i.e. principal names, alternative names, canonical names) in the implementation of Coq. First, the kernel of Coq that corresponds to the type checker of the base language only use principal names. Indeed, alternative names are just aliases for principal names, and thus computation (aka conversion) does not need to be aware of their existence. Identically, canonical names and Δ-equivalence are shortcuts for δ-equivalence, so it is not useful to integrate this notion in computation steps as δ-reduction is already a part of it. For his part, the user has access to both principal and alternative names depending on which namespace scope he uses. Finally, the proof writing machinery use all of them. For instance, tactics like auto or rewrite need to work on canonical names[7]. Whereas, notations need to work on alternative or principal name, since different notations, depending on the namespace scope, can be used for one logical object. We describe in the Chapter 5 our concrete solutions to provide this layered namespace.

[7] This is also the case for a large part of databases, like hints, type class instances..., and tools like search pattern

4.3 Inlining at functor application

In order to complete our toolbox that controls the δ relation of the base language, we introduce a new notion of inlining[8]. This notion allows to enforce the δ-reduction of a given qualified identifier. To illustrate the purpose of inlining, take the following functor:

```
Module Type OrderedType.
  Parameter t : Type.
  Parameter eq lt : t → t → Prop.
  Parameter eq_ refl : ∀x : t, eq x x.
  ...
  (* eq_ sym, eq_ trans, lt_ trans, lt_ strict *)
End OderedType.

Module SetList (X: OrderedType) .
  Definition elt := X.t.
  Definition t := list elt.
  Definition empty : t := nil.
  Definition is_ empty (l:t):= match l with ... .
  ...
  (* operations on set and their specification *)
End SetList.
```

Now, if we want to instantiate our functor *SetList* on Peano natural number (*nat*), which is a part of the standard library, we need to build a named module that implements *OrderedType*. Indeed, we have no notion of anonymous module as the one we can find in ML module systems. Hence, we need to duplicate the definitions, that are useful to instantiate the *SetList* functor, into a module as follows:

```
Module Type Nat_ OT.
  Definition t := nat.
  Definition eq := @eq nat.
  Definition lt :=lt_ nat.
  ...
End Nat_ OT.
```

[8] Based on experiments done by Claudio Sacerdoti Coen in the implementation of Coq.

In that context, if we instantiate our functor *SetList* to the module *Nat_OT*, the fields of the resulting structure will mention *Nat_OT.t*, *Nat_OT.eq* and *Nat_OT.lt*. However, it would have been preferable to have *nat*, and the standard notation for *eq* and *lt_nat* which are parts of the standard library. In other words, we need to have a way of inlining some definitions at functor applications. To do it, we add the keyword INLINE that specifies in the module parameter of a functor which fields are to inline. For instance, we write the module type *OrderedType* as follows:

MODULE TYPE *OrderedType*.
  PARAMETER INLINE t : *Type*.
  PARAMETER INLINE *eq lt* : $t \to t \to$ *Prop*.
  PARAMETER *eq_refl* : $\forall x : t, eq\, x\, x$.
  ...
END *OderedType*.

This extension is easy to set up in our module system. We need to add a mark on term parameter that are chosen to be inlined and we modify the STRUCT/APP (resp. MOD/APP) as follows:

STRUCT/APP

$$\frac{E\,|\,\mathcal{N} \vdash P : S_2 \qquad E\,|\,\mathcal{N} :: \# \vdash S_2^{\#}{}_{/P_2} \subseteq \{v/P\}S_1^{\#} \qquad \Delta = \mathit{collect}(P : S_2)}{E\,|\,\mathcal{N} \vdash \Delta \uparrow (\iota \circ \{v/P\}S) : \text{Struct}}$$

$$E\,|\,\mathcal{N} \vdash (v : S_1) \Rightarrow S : \text{Struct}$$

Where ι is the substitution $\{P.P_1.x_1/t_1, \ldots, P.P_n.x_n/t_n\}$ if and only if each field $v.P_i.x_i$ is marked inline in S_1, and for each $P.P_i.x_i$ we have $E\,|\,\mathcal{N} \vdash P.P_i.x_i \rhd_\delta t_i$.

Concluding remarks

As we have seen in this section, the inlining allows to remove intermediate names that are not always relevant for a development. The inlining is performed by enforcing the δ-reduction of chosen qualified identifiers. Together with the Δ-equivalence, they realize a toolbox that controls the δ-equivalence and eases the naming management for users.

Chapter 5

Implementation in Coq

In this chapter, we first give a short presentation of the module system of the last release of the Coq proof assistant (i.e. version 8.3). This system has been implemented as part of my Phd work and is mainly a subsystem of our new module system. Then, we present the implementation of our new module system formally described in Chapters 3 and 4. The status of this implementation is still experimental and is not completely finished, in particular for the namespace part.

5.1 A short presentation of the module system of Coq 8.3

The last release of the Coq proof assistant is the version 8.3. The module system provided by this release is a subsystem of our new one. It provides the unification of structures and signatures, an extended Include operator, the With operator [9], the Δ-equivalence, and the inlining at functor applications.

Syntax

More formally the syntax is the following:

[9] The With operator has been initially implemented by J. Chrząszcz and extended by C. Sacerdoti.

- Paths:

$P \quad := \quad Coq \mid P.p \mid v$

- Fields:

$e \quad := \quad P.x : T \mid P.x : T := t \mid$
$\qquad\quad P : S \mid P : S := S' \mid P = S$

- Structures:

$S \quad := \quad \langle e_1 \dots e_n \rangle \mid (v : S) \Rightarrow S'$

- Environments of declarations:

$E \quad := \quad \epsilon \mid E, e \mid E, (v : T)$

- Namespaces:

$\mathcal{N} \quad := \quad Coq \mid \mathcal{N} :: P$

An environment is composed of an environment of declarations and a namespace. On the contrary of the namespace notion defined in Chapter 3, the namespace part only represents paths of modules that are currently built interactively. All the rules of the class Env (cf. Chapter 3) use the head path of the namespace as qualifier in order to insert the object, considered within the rule, in the environment. In this version, the functors are considered generative, hence there is no Mod/App rule and the strengthening has no effect on higher-order structures (cf. Chapter 3). However, we have implemented an original include operator for higher-order structures. We define it by the following inference rule:

Struct/IncFun

$$\frac{\begin{array}{c} E \mid \mathcal{N} :: P \vdash \langle e_1 \dots e_n \rangle : \text{struct} \\ E \mid \mathcal{N} :: P \vdash (X_1 : S_1) \Rightarrow \dots \Rightarrow (X_n : S_n) \Rightarrow \langle e'_1 \dots e'_m \rangle : \text{struct} \\ \forall i \in [1 \dots n] \qquad E \mid \mathcal{N} :: P :: \# \vdash \langle e_1 \dots e_n \rangle^{\#} \subseteq \{X_{i=1 \dots i-1}/P\} S_i^{\#} \end{array}}{E \mid \mathcal{N} :: P \vdash \langle e_1 \dots e_n \{X_{i=1 \dots n}/P\} e'_1 \dots \{X_{i=1 \dots n}/P\} e'_m \rangle : \text{struct}}$$

Given a well-formed first-order structure and a well-formed higher-order structure under the path P, if the first-order structure fulfils the input of the higher-order structure, then we can combine the first-order structure together with the output structure of the higher-order one. This approach is quite unusual and is only possible in this version of our module system. Indeed, it does not have applicative functors and always derives structures in normal form. In that context the path substitution $\{X_i/P\}$ corresponds to the following set of substitutions at qualified identifiers

level:

$$\bigcup_{X_i.P'.x \in QI(\langle e_1 \ldots e_n \rangle)} \{X_i.P'.x / P.P'.x\}$$

This operator underlies the reformulation of both libraries Numbers and Structures[10], that are included in the standard library of Coq. Indeed, this include operator helps to solve sharing in diamond-like development. To illustrate it, take the following example where we encapsulate, within a module type, an axiomatization of a given theory, and where we build a functor that derives properties from this axiomatization:

```
Module Type AxTheory.
  Parameter T : Type.
  Parameter foo : T.
  Parameter bar : T → T → T .
End AxTheory.

Module BaseProp (X:AxTheory).
  ...
End BaseProp.
```

We continue our development with two functors that independently derives more properties from *BaseProp*:

```
Module Prop1 (X:AxTheory).
  Module Base := BaseProp X.
  ...
End Prop1.

Module Prop2 (X:AxTheory).
  Module Base := BaseProp X.
  ...
End Prop2.
```

[10] The modular library Numbers has been initially formalized by Evgeny Makarov, and reformulated by Pierre Letouzey and the library Structures has been written by Pierre Letouzey.

Finally, we group both functors into a single one:

```
Module Final (X:AxTheory).
  Module P1 := Prop1 X.
  Module P2 := Prop2 X.
  ...
End Final.
```

Due to the generative semantic of functors, if the modules *P1.Base* and *P2.Base* define inductive types or abstract fields, then the latter and the terms depending on them are not convertible. The first solution is to linearize the development. In other words, we instantiate the functor *Prop1* within the module *Prop2*, and we instantiate the functor *Prop2* within *Final*. The main drawbacks of this approach are the violation of the separation of concern, and the addition of a substantial numbers of imbricated modules. Indeed, in the previous version of the Numbers library, the modular structure of the final module had a depth of about 10 imbricated modules. Now, with the help of our single notion of structures and our extended include operator, we do as follows:

```
Module Type BaseProp (X:AxTheory).
  ...
End BaseProp.

Module Prop1 (X:AxTheory) (Base:BaseProp X).
  ...
End Prop1.

Module Prop2 (X:AxTheory) (Base:BaseProp X).
  ...
End Prop2.

Module Final (X:AxTheory).
  Include X <+ BaseProp <+ Prop1 <+ Prop2.
  ...
End Final.
```

First, we modify the functor *BaseProp* into a module type, without changing its content. Then, we parameterize *Prop1* and *Prop2* by *BaseProp X*, and we remove the *Base* submodule. Finally, we group everything in the final functor. The notation "<+" is syntactic sugar for a chain of include, and means that we first include X, then we include *BaseProp*... In the *Final* functor, the functor *BaseProp* is instantiated by the fields inherited from X. The functors *Prop1* and *Prop2* are both instantiated by the fields inherited from X and *BaseProp*.

Finally, this include operator can also be used to associate non-logical object such as notations, hints and tactics to a given structure. We give an example inspired from the standard library:

```
Module Type EqLt.
  Parameter t : Type.
  Parameter eq : t → t → Prop .
  Parameter lt : t → t → Prop .
End EqLt.

Module Type EqLtNotation (X:EqLt).
  Infix "==" := eq (at level 70, no associativity).
  Notation "x ~= y" := ( eq x y) (at level 70, no associativity).
  Infix "<" := E.lt.
  Notation "x > y" := (y<x) (only parsing).
  Notation "x < y < z" := (x<y ∧ y<z).
End EqLtNotation.
```

The higher-order structure *EqLtNotation* can be used to add notations to any structure or module that have the three fields *t*, *eq*, and *lt*:

```
Module Nat_OT' := Nat_Ot <+ EqLtNotation.
```

5.2 Implementation of the new system

The Coq proof assistant is written in Objective Caml. The system has about 180.000 lines of code organized in about 350 files. The source tree is divided into the following directories:

kernel: implementation of the type-checkers for both pCIC and module system. This is the critical part of the system and it represents 8% of the whole code.

library: interface of the global environment and management of both stack of objects for backtracking and synchronization of various tables of objects.

parsing: management of the abstract syntax trees and of pretty-printing.

interp: translation from syntax trees to untyped terms and untyped structure expressions.

pretyping: analysis of untyped terms.

proofs: management of interactive proofs.

tactics: implementation of predefined tactics and of the language of tactics.

toplevel: interactive loop.

checker: independent type-checker.

The implementation of our module system mostly concerns the kernel and the library. In the following we first describe our modifications in the kernel, and then those concerning the library.

5.2.1 In the kernel

The modules that are of interest for us are the following:

- `Names` defines the type of paths, constant names, inductive names and namespaces.
- `Entries` defines the type of untyped terms, structures and modules.
- `Declarations` defines the type of typed terms, structures and modules.
- `Mod_subst` defines types of path substitutions and Δ-equivalence.
- `Mod_Typing` implements the module and structure type-checking algorithm.
- `Modops` implements auxiliary functions on structures and modules such as strengthening and error reporting functions.
- `Subtyping` implements the subtyping algorithm.
- `Safe_Typing` is the interface between the kernel and the rest of the program. It defines the type of well-formed environments.

Names

Let us first describe the type of paths that is transparently defined in the module `Names`:

```
type module_path =
   | MPfile of dir_path
   | MPbound of mod_bound_id
```

```
| MPapp of module_path * module_path
| MPdot of module_path * label
```

Here, the constructor `MPfile` stands for a compilation unit path and a value of type `dir_path` is a list of names in reverse order. For instance, in the standard library the module `List` has the path `MPfile([List,Lists,Coq])`[11]. The constructor `MPbound` represents module path variables, `mod_bound_id` is the type of unique names: it is a name associated to an integer and it allows to generate fresh names. Finally, `MPapp` and `MPdot` are the regular constructors for module paths.

Then, we define the type `kernel_name` that represents the qualified identifiers:

```
type kernel_name = module_path * dir_path * label
```

It is opaquely defined and it is composed of a module path and a label, the `dir_path` part represents the list of opened sections. Since, modules can not be defined within sections, we will consider this part empty. Now, we define the type of qualified identifier for constants (i.e. term parameters and term definitions) and for mutual inductive type definitions:

```
type constant = kernel_name * kernel_name * kernel_name
```

```
type mutual_inductive = kernel_name * kernel_name * kernel_name
```

Both types are triplet of `kernel_name`, the first projection corresponds to the principal name, the second one to the canonical name (c.f. Chapter 4), and the third one is the alternative name considered by the user. Finally, a namespace is a set of valid paths and a map of `kernel_name`.

```
type namespace =MPset.t * kernel_name KNmap.t
```

Beside, the usual constructor and destructor functions for these types, we define three orders for the types `constant` and `mutual_inductive` together with the three corresponding instantiations of functors `Map.Make` and `Set.Make`. The first order is based on the principal name, it is mostly used for the kernel part of Coq. The second one is based on the canonical name, it is devoted to tables of non-logical objects. The third one is based on the alternative name. For instance, the `Map.Make` module instantiated with the principal name order is used to implement the environment of declarations, whereas the `Map.Make` module instantiated with the canonical name order is used, beyond others, in the implementation of the *auto* tactic.

[11] We use `Coq` as initial path in the implementation, whereas in our formalization we have used *Top*.

Entries

This module defines the types of untyped fields and untyped structure expressions. The type of fields is the following:

```
type specification_entry =
   | SPEconst of constant_entry
   | SPEmind of mutual_inductive_entry
   | SPEmodule of module_entry
   | SPEmodtype of structure_entry
```

A field is either a term definition (or parameter), a mutual inductive type definition, a module definition (or parameter), or a module type definition (i.e. a structure abbreviation). The type of untyped structure expressions is the following:

```
and struct_entry =
   | MSEident of module_path
   | MSEfunctor of mod_bound_id * module_struct_entry * struct_entry
   | MSEwith of struct_entry * with_declaration
   | MSEapply of struct_entry * struct_entry
   | MSEmerge of structure_entry list

and structure_entry = struct_entry * bool
```

A structure expression is either a path, a higher-order structure, a structure refinement, an application[12] or a merge of structures. The type `structure_entry` associates to a structure expression a boolean flag that is true if the considered expression is a module expression and false if it is a module type expression. Hence, a merge of structures can consider heterogeneous structure expressions. Finally, the type for untyped module definition is a pair of optional structures:

```
and module_entry =
   {mod_entry_type :  structure_entry option;
   mod_entry_expr :  structure_entry option}
```

If `mod_entry_type` is `None`, then it corresponds to an unsealed module definition. If `mod_entry_type` is `None`, then it corresponds to a module parameter.

[12] The application considered here is for module type application, the module path application is indeed included in the `MSEident` constructor.

Declarations

Now, we give the typed versions of fields and structure expressions as they are defined in the module `Declarations`:

```
type structure_field_body =
   | SFBconst of constant_body
   | SFBmind of mutual_inductive_body
   | SFBmodule of module_body
   | SFBmodtype of module_type_body

and structure_body = (label * structure_field_body) list

and struct_expr_body =
   | SEBident of module_path
   | SEBfunctor of mod_bound_id * module_type_body * struct_expr_body
   | SEBstruct of structure_body
   | SEBwith of struct_expr_body * with_declaration_body
```

A structure expression is either a path, a higher-order structure, a first-order structure or a structure refinement[13]. The bodies of a module or of a module type definition are defined as follows:

```
and module_body =
   {mod_mp :  module_path;
   mod_expr :  struct_expr_body option;
   mod_type :  struct_expr_body;
   mod_type_alg :  struct_expr_body option;
   mod_constraints :  constraints;
   mod_delta :  delta_resolver;
   mod_namespace :  namespace}

and module_type_body =
   {typ_mp :  module_path;
   typ_expr :  struct_expr_body;
   typ_expr_alg :  struct_expr_body option ;
   typ_constraints :  constraints;
   typ_delta :  delta_resolver;
   typ_namespace :  namespace}
```

[13]We keep the structure refinement construction for OCaml extraction purpose.

These types are very similar. The `mod_mp` (resp. `typ_mp`) field is the path of the definition. The `mod_expr` field is the optional implementation of the module, and the `typ_expr` field is the abbreviated structure. The `mod_type` field is the signature of the module definition. For both types we optionally keep an algebraic version of the structure expression for Ocaml extraction purpose. Finally, the inferred universe constraints, Δ-equivalence information and namespace are stored in their respective fields.

Mod_subst

This module defines path substitutions and the δ *toolbox* (aka Δ-equivalence and inlining). It also gives the support for lazy application of path substitutions. The type, that encodes our δ *toolbox*, is called `delta_resolver` and is defined as follows:

```
type delta_hint =
   | Inline of constr option
   | Equiv of kernel_name
   | Prefix_equiv of module_path

type delta_key =
   | KN of kernel_name
   | MP of module_path

type delta_resolver = delta_hint Deltamap.t
```

A value of type `delta_resolver` maps a value of type `delta_key` to a value of type `delta_hint`. The inlining is represented through the association of a kernel name with an optional Coq term (i.e. `constr`). The Δ-equivalence is represented either globally by mapping a path to another path, or, one by one, by mapping a kernel name to another kernel name. To illustrate the delta resolver, take the three following modules:

```
Module M.
  Parameter foo : T.
  Parameter bar : T.
End M.
```

```
Module P := M.

Module K.
  Include M.
  Parameter bla : T.
End K.
```

The module M has an empty delta resolver, the module P has a delta resolver that maps the module path P to the module path M, and the module K has a delta resolver that maps *K.foo* (resp. *K.bar*) to *M.foo* (resp. *M.bar*). Indeed, in the case of module inclusion (or module merging), we can not factorize the delta resolver since it would capture in our example the name *K.bla*.

Path substitutions are represented through a mapping from module path to a pair formed of a module path and a delta resolver:

```
type substitution = (module_path * delta_resolver) MPmap.t
```

In the codomain of a substitution, the domain of the delta resolver is intended to be prefixed by the module path. Hence, the delta resolver can be seen as an extension of the substitution. This extension is used to propagate the Δ-equivalence, and to perform inlining of constants.

Finally, the module `Mod_Subst` gives support for lazy application of path substitutions. It is is implemented through the following type and functions:

```
type 'a substituted =
   | LSval of 'a
   | LSlazy of substitution list * 'a
val from_val :  'a -> 'a substituted
val force :  (substitution -> 'a -> 'a) -> 'a substituted -> 'a
val subst_substituted :  substitution -> 'a substituted -> 'a substituted
```

For obvious performance reasons, the lazy application of path substitutions is used on Coq terms. The type `'a substituted` represents either a value of type `'a`, or a pair composed of a value and a list of substitutions. The function `subst_substituted` adds a substitution in the list of substitutions. The function `force` performs a sequential composition of the substitutions stored in the list, and applies it on the value. The sequential composition of substitutions is performed by the following function:

```
val join :  substitution -> substitution -> substitution
```

that depends on two other important functions that perform a substitution on the domain or codomain of a delta resolver:

```
val subst_dom_delta_resolver :
      substitution -> delta_resolver -> delta_resolver
val subst_codom_delta_resolver :
      substitution -> delta_resolver -> delta_resolver
```

Subtyping

This module implements the subtyping algorithm, formalized in our system by the rules of the class SUB, and it also implements the rules MERGE/FIELD/LEFT and MERGE/FIELD/RIGHT. The function for subtyping checks is :

```
val check_subtypes :  env -> module_type_body -> module_type_body -> constraints
```

As in the formalization, the subtyping algorithm is decomposed in five functions `check_*`, one for structures and the others for each kind of fields: module, module type, constant, inductive. If the subtype checking is successful then the function returns the set of constraints collected from each call of the conversion oracle ($\leq_{\beta\delta\iota\zeta}$). One can refer to [27] for information about the inference of universe constraints.

The merge rules are split in four functions `merge_*`, one for each kind of fields. They take two fields of the concerned kind, and return the most precise one and a set of constraints using the corresponding `check_*` function.

Mod_typing and Modops

These modules are devoted to the type checking of the module system. The core of the typing algorithm is in the module `Mod_typing`, this algorithm is derived from the one implemented in Ocaml [36], notably for the typing of path applications. The module `Modops` implements auxiliary functions that are nonetheless important for type-checking. We have the strengthening function and substitution functions over structures and fields, both of them are used for higher-order structure applications, and we have a function that perform both substitution and strengthening in parallel, this function implements a part of the rule STRUCT/PATH.

For type-checking concern, the module `Mod_typing` exports three functions, that are:

```
val translate_module :  env -> module_path -> bool -> module_entry
   -> module_body

val translate_module_type :  env -> module_path -> bool -> structure_entry
   -> module_type_body

val translate_struct_entry :  env -> module_path -> bool -> structure_entry
   -> struct_expr_body * struct_expr_body option *
      delta_resolver * namespace * Univ.constraints
```

These functions take as arguments an environment, a module path that correspond in our formalization to the head path of the namespace, a boolean which is a flag to block or authorize inlining and finally an untyped entry. They result is a typed declaration for module and mod-

ule type. The function `translate_struct_entry` returns the typed structure expression, which needs to be in normal form (i.e. either of the form `SEBstruct` or of the form `SEBfunctor`), an algebraic structure expression, and the inferred resolver, namespace and constraints.

The typing of a structure merging is performed by the private function:

```
val merge_structure_entry :  env -> module_path -> bool -> structure_entry list
```

The algorithm is the same as the one presented in Chapter 3, unless that it handle a n-ary structure merging. It proceeds as follows: first it translate the list of untyped structure expression using the function `translate_struct_entry`, then it merges structures of the obtained list from left to right. At each merge of structures step, it checks if the namespaces are coherent (i.e. overlapping indirections point to the same principal name), and then it builds the resulting structure by merging each subfields with the help of `Subtyping.merge_*` functions.

Safe_Typing

The interface between the kernel and the rest of Coq is the module `Safe_typing`. It defines the abstract type `safe_environment` which corresponds, in our formalization, to a well-formed environment. It provides functions that implement the rules of the class ENV (i.e. the rules for building well-formed environment). These functions are named `add_*`:

```
val add_constant :
   label -> constant_entry -> safe_environment ->
      constant * safe_environment

val add_mind :
   label -> mutual_inductive_entry -> safe_environment ->
      mutual_inductive * safe_environment

val add_module :
   label -> module_entry -> bool -> safe_environment ->
      module_path * delta_resolver * safe_environment

val add_modtype :
   label -> structure_entry -> bool -> safe_environment ->
      module_path * safe_environment
```

they all take as argument a label, an untyped entry, a safe environment, and return a safe environment. For modules we also return the delta resolver associated to the module, so that it can be propagated at the level of non-logical objects.

`Safe_Typing` also implements interactive modules and module types, this has been done by Jacek Chrząszcz [11]. The interactive building is implemented through the pair of functions `start_module` (resp. `start_modtype`) and `end_module` (resp. `end_modtype`). The function `start_module` corresponds to the rule ENV/NAMESPACE, and the function END_MODULE implements both STRUCT/ENV and ENV/MODDEF. For instance, take the following interactively built module:

`Coq <` MODULE *M.*
Interactive module M started.
`Coq <` DEFINITION $T := nat$.
T is defined.
`Coq <` DEFINITION $x := 0$.
x is defined.
`Coq <` END *M.*
M is defined.

We suppose that the environment before the interactive module is $E \,|\, \mathcal{N}$, and that we define the module M in the Top namespace. The command MODULE adds the path $Top.M$ in the environment: $E \,|\, \mathcal{N} :: Top.M$, then the definition are added with the `add_definition` function, they implicitly use the current path as qualifier:

$$E, (Top.M.T : Set := nat), (Top.M.x : nat := 0) \,|\, \mathcal{N} :: Top.M$$

The END command gathers the structure, here $\langle (Top.M.T : Set := nat), (Top.M.x : nat := 0) \,|\, \epsilon \rangle$, backtracks to the former environment, $E \,|\, \mathcal{N}$, and adds the module in this environment, $E(Top.M : S := S) \,|\, \mathcal{N}$ with S the considered structure. The mechanism of interactive module building has not change much since its original implementation. The backtracking mechanism and the stack of safe environment is well explained in [11] and hence we do not explain it again here. However, we have added supports for the δ resolver, and namespace. When calling the `start_module` function, we associate to the path an empty δ resolver value and an empty namespace value. While building the module we cumulate the inlining and the namespace informations given by the user, and Δ-equivalence inferred by the type-checker. When calling the `end_module` function, these values are packaged together with the structure value in a `module_body` value.

Our notion of structure allows us to perform an interesting modification in term of memory sharing. If a module built interactively is not sealed, then both implementation and signature of the module are given the same physical structure, this sharing is treated carefully and is stable under path substitutions.

Finally, we have added the basic include operator that works only for first order structures:

```
val add_include :  structure_entry -> bool -> safe_environment ->
      delta_resolver * safe_environment
```

This function uses `Mod_typing.translate_struct_entry` to translate the untyped structure entry, checks if the resulting structure is first-order and then adds each fields one by one in the current interactive module together with the inferred delta resolver and namespace.

5.2.2 Outside the kernel

An important part of the implementation of the module system is done outside of the kernel part of Coq. While the kernel implementation part deals with type-checking and logical objects packaging into modules, the library part of the implementation deals with non-logical objects packaging and sharing. To some extent, the implementation in the library part is similar to the one done in the kernel. Indeed, we have to provide for non-logical objects: path substitutions, merge, interactive and non-interactive module constructions... We first describe briefly the modules that are of interest for our implementation:

- `Libobject` and `Lib` defines the type of generic objects and manages the stack of objects.
- `Summary` registers the declaration of global tables, and keep them synchronized with respect to backtracks of the system.
- `Declaremods` implements the module operations on the non-logical features of Coq.
- `Library` implements the compilation and the load of module (vo files).
- `Nametab` manages the name visibility.
- `Global` contains the reference to a safe environment and re-export the `Safe_typing` interface.

The three modules `Libobject`, `Lib` and `Summary` do not changed much since their original adaptation for the module system done by Chrząszcz. This triptych handles through a stack of objects the backtracking mechanism [14]. The definition of generic objects is done in the `Libobject` module. The summary mechanism periodically stores the state of the system, in order to have an efficient backtracking mechanism. The implementation of these three modules are well explained in [11].

Declaremods

The module `Declaremods` implements a counterpart for the `Library` part of the implementation of the `Safe_typing.add_*`, `Safe_typing.start_*` and `Safe_typing.end_*` functions. When

[14] Corresponding to the *Back* and *Reset* commands

the user starts an interactive module (resp. module type) the function `Declaremods.start_module` is called. This function calls the `Global.start_module` in order to start the module in the global environment and the function `Lib.start_module` to put a special mark on top of the stack of objects and to store the state of the system. During the interactive module construction the objects defined by the user are pushed on top of the stack. When the module is ended, the function `Declaremods.end_module` is called. This function gather the objects defined within the module by invoking the function `Lib.end_module`. If the module is sealed by a structure expression then it retrieves the objects associated to it with the function `get_substobjs`. Then it calls the function `Global.end_module` to end the module in the environment and to retrieve the delta resolver associated to the module. Finally, the delta resolver information is propagated on all objects through an extended path substitution (cf. `Mod_Subst`) and objects are packaged into a module object pushed on top of the stack. The path substitution is the identity if the module is unsealed. If the module is sealed by a structure expression then the path substitution is defined as the mapping from the path associated to the structure expression to the path of the module.

For non interactive module the function `Declaremods.Declare_module` is called, the mechanism is similar to the one done in `End_module` except that the objects associated to the module are retrieved through the `get_substobjs` function for both module implementation and sealing.

The function `get_substobjs` allows to compute the objects associated to an untyped structure expression. To some extent it works as the function `Mod_typing.translate_struct_entry`. The objects associated to a module path are stored in the `modtab_substobjs` table. We have implemented an experimental `merge_objs` function in order to perform the merging of structure at non-logical objects level.

Finally, we have added support for the include operator, this done by the `Declaremods.add_include` function. Given the untyped structure entry, it retrieves the corresponding objects with the help of `get_substobjs`, substitutes them so that they are now associated to the current interactive module or module type, and finally packages them into an "include" object that is pushed on top of the stack.

Δ-equivalence in the proof machinery

We have seen earlier that we provide maps and sets implementations in the module `Names` based on canonical names. It allows us to provide table that work modulo δ-equivalence. Indeed, given a δ class of names, non-logical objects associated to the whole class are stored only for the canonical name. However this is not sufficient for the proof machinery to work transparently over name issues. We have adapted the implementation of discrimination nets used for the *auto* tactics so that it works modulo Δ-equivalence. We have also adapted some approximations of terms, such as `constr_pattern`, so that for a given term its approximation only use canonical names. Hence when a tactic searches for a pattern in a given term, this is made modulo Δ-equivalence without modifying the implementation of the tactic.

5.3 User front end

In this Section, we describe how users interact with the new module system.

5.3.1 Impact of the structure-based system

The unification of module implementations and module types allows a more liberal use of the system. Indeed, modules type are not anymore only considered as interfaces, they are now structure abbreviations. On the contrary of the previous module system, lemmas and theorem can now be proven interactively within module types. For instance, take the following module type that defines the inductive type of list together with functions and lemmas related to it:

```
Module Type MyListTheory.
  Inductive List (A : Set) : Set :=
    | nil : List A
    | cons : A → List A → List A.
  Fixpoint foo : ...
  Lemma bar : ...
  ...
End MyListTheory.
```

At this point, our theory of list is not declared as a module and hence the components of this theory are not projectible. However, it is quite simple to declare it, since structure abbreviations can be used as module implementations:

```
Module Lists := MyListTheory.
```

Now, because of strengthening, all copies of the *Lists* module will create new convertible instances of *Lists* and all non-logical objects, such as hints, associated to one of these instances will be shared among all instances. On the other hand , if we want to create a new instance of our list theory, that is not convertible to the *Lists* one and that does not share non-logical objects, then we simply redeclare *MyListTheory* as a new module.

This approach is more interesting for higher-order structures. To some extent, it allows us to have both generative and applicative functors. For instance, take the following functorial module type:

```
Module Type MyTheory( X : S ).
  Inductive T : Set :=
    | C1 : T
    | C2 : T →T
  ...
End MyTheory.
```

We can declare this high-order structure within a module, and given a module M that implements S we can instantiate the resulting functor:

```
Module P := MyTheory.
Module P1 := P M.
Module P2 := P M.
Module P3 := MyTheory M.
```

Here, we have that *P1.C1* is convertible to *P2.C1* because, thanks to the strengthening, both reduces to *(P M).C1*. However, the module *P3* declares a new inductive type, and hence *P3.C1* is not convertible to both *P1.C1* and *P2.C1*.

Reciprocally, module expressions can be used as module types. It allows to extract the principal signature of a module expression and binds it to a structure abbreviation. With the help of the Include operator, it can also be used to add sharing constraints within a module type.

5.3.2 Merge of structures

The merge of structure operator offers new ways to combine structures. Coq provides a n-ary merge of structure operator. It can be given an arbitrary mixture of module and module type expressions.

5.3.3 The δ-toolbox

The δ-toolbox is composed of the Δ-equivalence and the inlining. The Δ-equivalence works transparently for the user, it permits to handle the sharing of non-logical objects and it allows tactics to work transparently other name equivalence issues. However, the inlining is directed by the user, he has to choose which fields we want to be inlined at functor applications and he can eventually prevent inlining by prefixing the application with the symbol !.

5.3.4 Namespaces

Namespaces are not yet fully implemented in Coq, however we give here the basic constructions and some example of use. There is two kinds of namespaces: we have the global namespace that correspond to top-level paths and theirs indirections, and local namespaces that are defined within modules.

Global Namespaces

Global namespaces are used to give an alternative name management to the one defined by the modular hierarchy of a development. For instance, the Coq standard library could be equipped with a global namespace containing the paths, *Lists*, *Natural*, *Integer*... and this namespace could collect with the help of indirection the main definitions and lemmas stated for each theory in the standard library. Then in other developments this namespace could be extended giving the possibility to have a unique qualifier for objects of the same concern. For instance, we can imagine to have a special preamble file devoted to the declaration of the global namespace as follows:

(File preamble.ns *)*

NAMESPACE *Foo.*
NAMESPACE *Foo.Bar.*
EXTENDS *Coq.List.*
EXTENDS *OtherDev.Theory.*

We could add new valid qualifiers in the global namespace, and extend some existing one. Of course a such new feature in Coq asks a lot of engineering to be able to solve conflicting indirections in diamond-like development.

Local Namespaces

A local namespaces is a namespaces defined within a module, it can not be extended with indirections outside of the scope of the module. It can be used to provide qualified names to some components of a module without defining a submodule. For instance, they can be handy to provide naming scope for Coq records which are basically inductive types:

NAMESPACE *Group.*
  RECORD *G (M : Type) : Type :=make_g {*

op : $M \to M \to M$; *id* : M;
assoc :...;
inv : ...; }.
END *Group*.

Now, the *Group* qualifier can be used in the current module, to qualify the name of lemmas and definitions that are related to groups.

LEMMA *unique_inv* IN *Group* : ....

We could also think to use them for collecting automatically generated definitions and lemmas such as inductive elimination schemes and proof obligations.

5.4 Concluding remarks

We have presented the module system of the latest release of Coq, and we have described the on-going implementation for the module system presented in this dissertation. The module system of Coq v8.3 has been used to reformulate a part of the standard library[15], and it has been admitted that the extended include operator and the structure based system was useful for it. On the implementation side, the remaining work is to integrate namespaces and applicative functors outside of the kernel and to provide support for global namespaces as sketched in the Section 5.3.

[15] FSets, Structures, Numbers and a part of ZArith.

Conclusion

We have presented different proof assistants together with their respective solutions to manage theories. We have seen that module system is a well adapted solution for theory management.

We have presented a new module system for the Coq proof assistant. The originality of this work resides in our new approach to manage the name-space and to combine structures. While keeping some ML module system features, we claim that this new module system is more adequate for theory management. Indeed, the namespace system allows the user to have different naming views on a development, and the structure-based system is more handy and more liberal for combining theory in a step by step fashion.

We have proved, through an incremental construction, that our module system is a conservative extension of the underlying base language. The steps considered in this construction are the following: we have defined first a trivial extension of the base language by adding the possibility of qualifying declarations in the environment, and we have considered the possibility of giving alternative names to such declarations. We have called this system $\mathcal{B}_{\mathcal{I}}$. We have proved that alternative names can be removed from an arbitrary derivation. The second system, called $\mathcal{M}$, is an extension of $\mathcal{B}_{\mathcal{I}}$, where we have added a new notion of structures. In that context, we are able to recover the notion of module by associating a sub-namespace and a pair of structures. Then, we have investigated two admissible extensions of $\mathcal{M}$, the first one simulates module renaming and the other one realizes the operator of structure merging. We have shown that a derivation in the system $\mathcal{M}$ can be translated to a derivation in the system $\mathcal{B}_{\mathcal{I}}$ with the same conclusion by flattening modules. Finally in the last system, called $\mathcal{HM}$, we have extended $\mathcal{M}$ with higher-order structures and applicative functors. We have proved that the δ-reduction is weakly normalizing and we have given the algorithm extracted from this proof. This result was necessary to eliminate path applications in terms, and hence to prove the conservativity of this last extension with respect to $\mathcal{M}$.

Moreover, we have introduced a more technical extension of our system, that we have informally called the *δ-toolbox*. This extension allows to have a simplified control over the sharing constraints implied by modular constructions, and to select fields that need to be unfolded at functor application.

Lastly, we have presented the module system that we have implemented for the latest release of Coq, and we have described the on-going implementation for the module system presented in this dissertation. The priority work is of course to finalize this implementation and to perform

some extensive testing. But we can already say that the new namespace notion offers some interesting perspectives in term of new features for the Coq proof assistant. For instance, we could allow module renaming through the use of namespace. A such feature is useful to lighten modular development where some modules are duplicated only to give them a more adequate name. This extension would be easily implementable and would not modify our formalization since it could be performed by renaming each sub-fields of the module.

We could also extend namespaces to higher-order namespaces. By higher-order namespaces, we mean to define paths parameterized by path variables, that can produce regular paths at instantiation. As an illustration, take a development which purpose is to formalize a theory parameterized over a given abstract specification of a theory. Such a development ends up with a functor, and clients of this development can instantiate the final functor in order to get the concrete theory. This corresponds for example to the design of the Numbers development of the standard library of Coq. The notion of higher-order namespace would allow to define alternative name-spaces that depend on the argument of the functor. Hence, when instantiating the functor with a concrete module, a namespace coherent with the module argument would be created and would be filled with the different logical and non-logical objects that are outputted by the functor. It would give a process of contextual name generation.

Finally, it could be interesting to investigate the unification of namespace and the section mechanism of Coq. This could allow to have local contexts associated to valid paths of the namespace. To some extent, this new construction would act as the original notion of *Locales* in Isabelle (i.e. without interpretation and development graph) and could be an alternative to the use of modules in some cases.

Appendix A: Notes on the ML module system

The idea of giving a module system for ML arise in the early 80's. MacQueen [39] gives a first description of such a module system, introducing the basic concept of structures, signatures and functors. He axes his research on inheritance and sharing. The approach taken by SML'90, has been formally studied by Macqueen [40] and Harpper *et al.* [26, 25]. They advocate the use of dependant type to model dependency and parameterization. Structures are tuples, paths are projections, functors are lambda abstractions and signatures corresponds to product types and strong sum types. While this approach answers several modular aspects of programming (propagation of type equations, the dot notation, and sub-structure and functor dependencies), it does not support type abstraction. Indeed, in SML'90 the sealing is transparent, that is, sealing a module with signature does not hide the definition of any visible type definition.

At the same time, Mitchell and Plotkin [43] use existential types to give an account for abstract data types. Existential types provide a logical foundation for type abstraction, and hence allow an opaque approach for data abstraction. However, a value of existential type is not as flexible as a classical ML module. Indeed, in order to use a value v of type $\exists\alpha.T$, one must "open" the value as `open` v `as` $[\alpha, x]$ `in` u, since there is no dot notation as in ML module system. In that example, the scope of the abstract type α and of the associated function x (of type T) is restricted to u. On the other side, the scope of values and types provided by a module is program wide. Finally, both approaches have their limits, the transparent approach propagates too much type information loosing any hope in data abstraction, whereas the opaque approach is too much restrictive and hence prevent any form of sharing.

The dichotomy between the opaque approach and the transparent approach has been solved, in the 90's, with the notion of translucent signature and opaque sealing, independently given by Harpper and Lillibridge [24] and Leroy [34]. In both work, the notion of signature is enriched so that a signature can contain both abstract and manifest type components. However, these two works differ in the design of the module system. Leroy's modules are *second-class*, in the sense that the module language exists on a separate plane from the "core" language. Whereas, Harpper and Lillibridge's modules are *first-class*, they do not make any distinction between the "core" and the module languages, this approach make the type-checking undecidable. Both module systems support higher-order functors, however the propagation of type informations is not satisfying for

them. For instance, take the following functors written in SML syntax:

```
signature SIG = sig type t ...  end

  functor Apply (F : SIG -> SIG) (X : SIG) = F(X)
  functor Ident (X : SIG) = X
  structure Arg :  SIG = struct type t = int ...  end

  structure Res1 = Apply(Ident)(Arg)
  (* Res1.t != Arg.t *)
  structure Res2 = Ident(Arg)
  (* Res2.t = Arg.t *)
```

`Apply` takes a functor argument `F` of signature `SIG -> SIG` and a structure argument `X` of signature `SIG` and it applies `F` to `X`. Ideally, `Apply(F)(X)` should be semantically indistinguishable from `F(X)`. However, this turns out not to be the case, in both systems. In fact, the type of the structure outputted by the functor `Apply` is `SIG`. Hence , if we apply `Apply` to the identity functor `Ident`, and to the structure `Arg`, then the result `Res1` has type `SIG` as well, giving us no indication that its type component `t` is in fact equal to `Arg.t`.

One approach to remedying this problem was proposed by MacQueen and Tofte [41] and incorporated into the SML/NJ compiler. Their solution is to "re-type-check" the body of the `Apply` functor at every application, exploiting knowledge of `Apply`'s actual arguments to propagate more type informations. However, in the context of separate compilation, it is inapplicable, as `Apply`'s implementation may not be available. A more satisfying solution has been proposed by Leroy in [35]. He proposes an "applicative" semantics for functors as an alternative to Standard ML's "generative" semantics. It allows to give fully transparent signatures for higher-order functors. In Leroy's formalism, the syntactic class of paths is extended with path applications, and functor applications appearing in type paths must be in named form. Hence in the previous example, the output signature of the functor `Apply` is `sig type t= F(X).t ...  end`. However, in the general case when we allow functor applications to anonymous module as it is the case in OCaml, the module system does not have anymore the principal type property [36]. More recently, Russo [49] designed a type system that supports applicative functors and that allows modules to be used as first-class values. Instead of amalgamating the features of both module and core in a single language, he provides constructs for packing module values as core values and opening core values as module values, allowing programs to alternate between modules and core level computation.

Appendix B: The full system

1 Syntax

$$
\begin{array}{lcl}
P & := & Top \mid v \mid P.p \mid (P\,P) \\
t, u, T, U & := & v \mid P.x \mid \lambda v : T.t \mid \forall v : T.U \mid (t\,u) \mid s \\
\mathcal{N} & := & Top[\mathcal{I}] \mid \mathcal{N} :: P[\mathcal{I}] \\
\mathcal{I} & := & \epsilon \mid \mathcal{I}, (P.x \leftarrow P.x) \\
e & := & P.x : T \mid P.x : T := t \mid \\
 & & P : S \mid P : S := S' \mid (P = S) \\
S & := & \langle e_1 \ldots e_n, \mathcal{N} \rangle \mid (v : S) \Rightarrow S \\
E & := & \epsilon \mid E, e \mid E, (v : T) \mid E, (v : S)
\end{array}
$$

2 Judgements

$E \mid \mathcal{N} \vdash t : T$	
$E \mid \mathcal{N} \vdash t_1 =_{\beta\delta} t_2$	
$E \mid \mathcal{N} \vdash T_1 \leq_{\beta\delta} T_2$	
$E \mid \mathcal{N} \vdash S : \text{struct}$	The structure S is well-formed.
$E \mid \mathcal{N} \vdash P : S$	The module P has type S.
$E \mid \mathcal{N} \vdash S_1 \uplus S_2 \rightsquigarrow S_3$	The merge of S_1 and S_2 yields S_3.
$E \mid \mathcal{N} \vdash e_1 \uplus e_2 \rightsquigarrow e_3$	The merge of e_1 and e_2 yields e_3.
$E \mid \mathcal{N} \vdash S_1 \subseteq S_2$	The structure S_1 is a subtype of S_2
$E \mid \mathcal{N} \vdash e_1 \subseteq e_2$	The field e_1 is a subtype of e_2

3 Typing rules

- The Term class rules derive judgements of the form $E \,|\, \mathcal{N} \vdash t : T$

Term/Ax
$$\frac{E \,|\, \mathcal{N} \vdash \text{ok} \qquad (s_1, s_2) \in Ax}{E \,|\, \mathcal{N} \vdash s_1 : s_2}$$

Term/App
$$\frac{E \,|\, \mathcal{N} \vdash t : \forall v : U, T \qquad E \,|\, \mathcal{N} \vdash u : U}{E \,|\, \mathcal{N} \vdash (t\, u) : \{u/v\}T}$$

Term/Lambda
$$\frac{E \,|\, \mathcal{N} \vdash \forall v : T, U : s \qquad E, (v : T) \,|\, \mathcal{N} \vdash t : U}{E \,|\, \mathcal{N} \vdash \lambda v : T, t : \forall v : T, U}$$

Term/Prod
$$\frac{E \,|\, \mathcal{N} \vdash T : s_1 \qquad (s_1, s_2, s_3) \in \mathit{Prod} \qquad E, (v : T) \,|\, \mathcal{N} \vdash U : s_2}{E \,|\, \mathcal{N} \vdash \forall v : T, U : s_3}$$

Term/Sub
$$\frac{E \,|\, \mathcal{N} \vdash U : s \qquad E \,|\, \mathcal{N} \vdash t : T \qquad E \,|\, \mathcal{N} \vdash T \leq_{\beta\delta} U}{E \,|\, \mathcal{N} \vdash t : U}$$

Term/Acc
$$\frac{E \,|\, \mathcal{N} \vdash \text{ok} \qquad \mathcal{N}(P'.x') = P.x \qquad (P.x : T) \in E}{E \,|\, \mathcal{N} \vdash P'.x' : T}$$

Term/Var
$$\frac{E \,|\, \mathcal{N} \vdash \text{ok} \qquad (v : T) \in E}{E \,|\, \mathcal{N} \vdash v : T}$$

Term/Field
$$\frac{E \,|\, \mathcal{N} \vdash P : \langle e_1 \ldots e_n \,|\, \mathcal{N}_P \rangle \qquad \mathcal{N}_P(P.P'.x) = P.P''.x \qquad (P.P''.x : T) \in \langle e_1 \ldots e_n \rangle}{E \,|\, \mathcal{N} \vdash P.P'.x : T}$$

Term/FieldInd
$$\frac{E \,|\, \mathcal{N} \vdash P : \langle e_1 \ldots e_n \,|\, \mathcal{N}_P \rangle \qquad \mathcal{N}(P'.x) = P.P''.x \qquad (P.P''.x : T) \in \langle e_1 \ldots e_n \rangle}{E \,|\, \mathcal{N} \vdash P'.x : T}$$

- The Env class rules derive judgements of the form $E \,|\, \mathcal{N} \vdash \text{ok}$

$$\textsc{Env/Empty}\quad \frac{}{\epsilon \,|\, Top \vdash \text{ok}} \qquad \textsc{Env/Var}\quad \frac{E \,|\, \mathcal{N} \vdash \text{ok} \qquad E \,|\, \mathcal{N} \vdash T : s \qquad s \in S}{E, (v : T) \,|\, \mathcal{N} \vdash \text{ok}}$$

$$\textsc{Env/Par}\quad \frac{E \,|\, \mathcal{N} \vdash \text{ok} \qquad E \,|\, \mathcal{N} \vdash T : s \qquad s \in S \qquad P \in \mathcal{N}}{E, (P.x : T) \,|\, \mathcal{N} \vdash \text{ok}}$$

$$\textsc{Env/Def}\quad \frac{E \,|\, \mathcal{N} \vdash \text{ok} \qquad E \,|\, \mathcal{N} \vdash t : T \qquad P \in \mathcal{N}}{E, (P.x : T := t) \,|\, \mathcal{N} \vdash \text{ok}}$$

$$\textsc{Env/ModVar}\quad \frac{E \,|\, \mathcal{N} :: P \vdash \text{ok} \qquad E \,|\, \mathcal{N} :: P :: v \vdash S : \text{struct}}{E, (v : S) \,|\, \mathcal{N} :: (P\, v) \vdash \text{ok}}$$

$$\textsc{Env/ModPar}\quad \frac{E \,|\, \mathcal{N} :: P \vdash \text{ok} \qquad E \,|\, \mathcal{N} :: P \vdash S : \text{struct}}{E, (P : S) \,|\, \mathcal{N} \vdash \text{ok}}$$

$$\textsc{Env/ModDef}\quad \frac{\begin{array}{c} E \,|\, \mathcal{N} :: P \vdash \text{ok} \\ E \,|\, \mathcal{N} :: P \vdash S_1 : \text{struct} \qquad E \,|\, \mathcal{N} :: P \vdash S_2 : \text{struct} \qquad E \,|\, \mathcal{N} :: P \vdash S_1 \subseteq S_2 \end{array}}{E, (P : S_2 := S_1) \,|\, \mathcal{N} \vdash \text{ok}}$$

$$\textsc{Env/Namespace}\quad \frac{E \,|\, \mathcal{N} \vdash \text{ok} \qquad P \in \mathcal{N} \qquad P.p \notin \mathcal{N}}{E \,|\, \mathcal{N} :: P.p \vdash \text{ok}} \qquad \textsc{Env/Indirect}\quad \frac{E \,|\, \mathcal{N} \vdash \text{ok} \qquad P.x \in E \qquad P' \in N}{E \,|\, \mathcal{N} \cup (P.x \leftarrow P'.x') \vdash \text{ok}}$$

- The Struct class rules derive judgements of the form $E \,|\, \mathcal{N} \vdash S : \text{struct}$

$$\textsc{Struct/Env}\quad \frac{E, e_1, \ldots, e_n, \,|\, \mathcal{N} :: P :: \mathcal{N}_P \vdash \text{ok} \qquad \mathit{Prefix}(P, \langle e_1 \ldots e_n \,|\, \mathcal{N}_P \rangle)}{E \,|\, \mathcal{N} :: P \vdash \langle e_1 \ldots e_n \,|\, \mathcal{N}_P \rangle : \text{struct}}$$

$$\textsc{Struct/Fun}\quad \frac{E, (v : S), \mathcal{N} :: (P\, v) \vdash S_1 : \text{struct}}{E \,|\, \mathcal{N} :: P \vdash (v : S) \Rightarrow S_1 : \text{struct}}$$

$$\textsc{Struct/Merge}\quad \frac{E \,|\, \mathcal{N} :: P \vdash \langle e_1 \ldots e_n \,|\, \mathcal{N}_P \rangle : \text{struct} \qquad E \,|\, \mathcal{N} :: P \vdash \langle e'_1 \ldots e'_m \,|\, \mathcal{N}'_P \rangle : \text{struct} \qquad E \,|\, \mathcal{N} :: P \vdash \langle e_1 \ldots e_n \,|\, \mathcal{N}_P \rangle \uplus \langle e'_1 \ldots e'_m, \mathcal{N}'_P \rangle \rightsquigarrow \langle e''_1 \ldots e''_k, \mathcal{N}''_P \rangle}{E \,|\, \mathcal{N} :: P \vdash \langle e''_1 \ldots e''_k \,|\, \mathcal{N}''_P \rangle : \text{struct}}$$

$$\textsc{Struct/App}\quad \frac{E \,|\, \mathcal{N} \vdash (v : S_1) \Rightarrow S : \text{struct} \qquad E \,|\, \mathcal{N} \vdash P : S_2 \qquad E \,|\, \mathcal{N} :: \# \vdash S_2^{\#}{}_{/P} \subseteq S_1^{\#}}{E \,|\, \mathcal{N} \vdash \{v/P\}S : \text{struct}}$$

$$\textsc{Struct/Path}\quad \frac{E \,|\, \mathcal{N} :: P \vdash P' : S}{E \,|\, \mathcal{N} :: P \vdash (\{P/P'\}S)_{/P'} : \text{struct}}$$

- The Mod class rules derive judgements of the form $E \,|\, \mathcal{N} \vdash P : S$

$$\textsc{Mod/Acc}\quad \frac{E \,|\, \mathcal{N} \vdash \text{ok} \qquad (P : S) \in E}{E \,|\, \mathcal{N} \vdash P : S}$$

$$\textsc{Mod/Field}\quad \frac{E \,|\, \mathcal{N} \vdash P : \langle e_1 \ldots e_n \,|\, \mathcal{N}_P \rangle \qquad (P.P' : S) \in \langle e_1 \ldots e_n \,|\, \mathcal{N}_p \rangle}{E \,|\, \mathcal{N} \vdash P.P' : S}$$

$$\textsc{Mod/Sub}\quad \frac{E \,|\, \mathcal{N} \vdash P : S \qquad E \,|\, \mathcal{N} :: \# \vdash S^{\#} \subseteq S_1^{\#}}{E \,|\, \mathcal{N} \vdash P : S_1}$$

$$\textsc{Mod/App}\quad \frac{E \,|\, \mathcal{N} \vdash P_1 : (v : S_1) \Rightarrow S \qquad E \vdash P_2 : S_2 \qquad E \,|\, \mathcal{N} :: \# \vdash S_2^{\#}{}_{/P_2} \subseteq S_1^{\#}}{E \,|\, \mathcal{N} \vdash (P_1\, P_2) : \{v/P_2\}S}$$

- The MERGE class rules derive judgements of the form $E \,|\, \mathcal{N} \vdash S_1 \ \uplus \ S_2 \rightsquigarrow S_3$ or $E \,|\, \mathcal{N} \vdash e_1 \ \uplus \ e_2 \rightsquigarrow e_3$

MERGE/EMPTY

$$\frac{}{E \,|\, \mathcal{N} \vdash \langle \epsilon \,|\, \mathcal{N}_1 \rangle \ \uplus \ \langle \epsilon \,|\, \mathcal{N}_2 \rangle \rightsquigarrow \langle \epsilon \,|\, \mathcal{N}_1 \cup \mathcal{N}_2 \rangle}$$

MERGE/LEFT

$$\frac{\begin{array}{c} PathCond(e_1 \,|\, \mathcal{I}_{e_1}, S) \\ E, e_1 \,|\, \mathcal{N} + \mathcal{I}_{e_1} \vdash \langle e_2 \ldots e_n \,|\, \mathcal{N}_1 \rangle \ \uplus \ S \rightsquigarrow \langle e_2'' \ldots e_k'' \,|\, \mathcal{N}_3 \rangle \end{array}}{E \,|\, \mathcal{N} \vdash \langle e_1 \ldots e_n \,|\, \mathcal{N}_1 + \mathcal{I}_{e_1} \rangle \ \uplus \ S \rightsquigarrow \langle e_1 \ldots e_k'' \,|\, \mathcal{N}_3 + \mathcal{I}_{e_1} \rangle}$$

MERGE/RIGHT

$$\frac{\begin{array}{c} PathCond(e_1' \,|\, \mathcal{I}_{e_1'}, S) \\ E, e_1' \,|\, \mathcal{N} + \mathcal{I}_{e_1'} \vdash S \ \uplus \ \langle e_2' \ldots e_m' \,|\, \mathcal{N}_2 \rangle \rightsquigarrow \langle e_2'' \ldots e_k'' \,|\, \mathcal{N}_3 \rangle \end{array}}{E \,|\, \mathcal{N} \vdash S \ \uplus \ \langle e_1' \ldots e_m' \,|\, \mathcal{N}_2 + \mathcal{I}_{e_1'} \rangle \rightsquigarrow \langle e_1' \ldots e_k'' \,|\, \mathcal{N}_3 + \mathcal{I}_{e_1'} \rangle}$$

MERGE/MATCH

$$\frac{\begin{array}{c} name(e_1) = name(e_1') \qquad \mathcal{I}_{e_1''} = \mathcal{I}_{e_1} \cup \mathcal{I}_{e_1'} \qquad E \,|\, \mathcal{N} \vdash e_1 \ \uplus \ e_1' \rightsquigarrow e_1'' \\ E, e_1'' \,|\, \mathcal{N} + \mathcal{I}_{e_1''} \vdash \langle e_2 \ldots e_n \,|\, \mathcal{N}_1 \rangle \ \uplus \ \langle e_2' \ldots e_m' \,|\, \mathcal{N}_2 \rangle \rightsquigarrow \langle e_2'' \ldots e_k'' \,|\, \mathcal{N}_3 \rangle \end{array}}{E \,|\, \mathcal{N} \vdash \langle e_1 \ldots e_n \,|\, \mathcal{N}_1 + \mathcal{I}_{e_1} \rangle \ \uplus \ \langle e_1' \ldots e_m' \,|\, \mathcal{N}_2 + \mathcal{I}_{e_1'} \rangle \rightsquigarrow \langle e_1'' \ldots e_k'' \,|\, \mathcal{N}_3 + \mathcal{I}_{e_1''} \rangle}$$

MERGE/FIELD/RIGHT

$$\frac{E \,|\, \mathcal{N} \vdash e_2 \subseteq e_1}{E \,|\, \mathcal{N} \vdash e_1 \ \uplus \ e_2 \rightsquigarrow e_2}$$

MERGE/FIELD/LEFT

$$\frac{E \,|\, \mathcal{N} \vdash e_1 \subseteq e_2}{E \,|\, \mathcal{N} \vdash e_1 \ \uplus \ e_2 \rightsquigarrow e_1}$$

- The Sub class rules derive judgements of the form $E \,|\, \mathcal{N} \vdash S_1 \subseteq S_2$ or $E \,|\, \mathcal{N} \vdash e_1 \subseteq e_2$

Sub/Struct

$$\frac{E \,|\, \mathcal{N} \vdash \langle e_1 \dots e_n \,|\, \mathcal{N}_1 \rangle : \text{struct} \qquad E \,|\, \mathcal{N} \vdash \langle e'_1 \dots e'_m \,|\, \mathcal{N}_2 \rangle : \text{struct} \qquad \mathcal{N}_2 \subseteq \mathcal{N}_1 \quad \phi : [1 \dots m] \mapsto [1 \dots n] \qquad \forall i \in [1 \dots m] \qquad E, e_1, \dots, e_n \,|\, \mathcal{N} :: \mathcal{N} \vdash e_{\phi(i)} \subseteq e'_i}{E \,|\, \mathcal{N} \vdash \langle e_1 \dots e_n \,|\, \mathcal{N}_1 \rangle \subseteq \langle e'_1 \dots e'_m \,|\, \mathcal{N}_2 \rangle}$$

Sub/Fun

$$\frac{E, \mathcal{N} :: P :: v \vdash S' \subseteq S \qquad E, (v : S'), \mathcal{N} :: (P\, v) \vdash S_1 \subseteq S'_1}{E \,|\, \mathcal{N} :: P \vdash (v : S) \Rightarrow S_1 \subseteq (v : S') \Rightarrow S'_1}$$

Sub/Def/Def

$$\frac{E \,|\, \mathcal{N} \vdash t =_{\beta\delta} u \qquad E \,|\, \mathcal{N} \vdash T \leq_{\beta\delta} U}{E \,|\, \mathcal{N} \vdash (P.x : T := t) \subseteq (P.x : U := u)}$$

Sub/Par/Par

$$\frac{E \,|\, \mathcal{N} \vdash T \leq_{\beta\delta} U}{E \,|\, \mathcal{N} \vdash (P.x : T) \subseteq (P.x : U)}$$

Sub/Def/Par

$$\frac{E \vdash T \leq_{\beta\delta} U}{E \,|\, \mathcal{N} \vdash (P.x : T := t) \subseteq (P.x : U)}$$

Sub/ModD/ModP

$$\frac{E \,|\, \mathcal{N} :: \# \vdash S_1^{\#} \subseteq S_3^{\#}}{E \,|\, \mathcal{N} \vdash (P : S_1 := S_2) \subseteq (P : S_3)}$$

Sub/ModD/ModD

$$\frac{E \,|\, \mathcal{N} :: \# \vdash S_1^{\#} \subseteq S_3^{\#} \qquad E \,|\, \mathcal{N} :: \# \vdash S_2^{\#} \subseteq S_4^{\#} \qquad E \,|\, \mathcal{N} :: \# \vdash S_4^{\#} \subseteq S_2^{\#}}{E \,|\, \mathcal{N} \vdash (P : S_1 := S_2) \subseteq (P : S_3 := S_4)}$$

Sub/ModP/ModP

$$\frac{E \,|\, \mathcal{N} :: \# \vdash S_1^{\#} \subseteq S_2^{\#}}{E \,|\, \mathcal{N} \vdash (P : S_1) \subseteq (P : S_2)}$$

- The DELTA class rules defines the δ reduction relation:

DELTA/DEF

$$\frac{E \,|\, \mathcal{N} \vdash \text{ok} \qquad \mathcal{N}(P'.x') = P.x \qquad (P.x : T := t) \in E}{E \,|\, \mathcal{N} \vdash P'.x' \rhd_\delta t}$$

DELTA/PAR

$$\frac{E \,|\, \mathcal{N} \vdash \text{ok} \qquad \mathcal{N}(P'.x') = P.x \qquad P'.x' \neq P.x \qquad (P.x : T) \in E}{E \,|\, \mathcal{N} \vdash P'.x' \rhd_\delta P.x}$$

DELTA/FIELDDEF

$$\frac{\begin{array}{c} E \,|\, \mathcal{N} \vdash P : \langle e_1 \dots e_n \,|\, \mathcal{N}_P \rangle \\ (\mathcal{N} \cup \mathcal{N}_P)(P''.x) = P.P'.x \qquad (P.P'.x : T := t) \in \langle e_1 \dots e_n \rangle \end{array}}{E \,|\, \mathcal{N} \vdash P''.x \rhd_\delta t}$$

DELTA/FIELDPAR

$$\frac{\begin{array}{c} E \,|\, \mathcal{N} \vdash P : \langle e_1 \dots e_n \,|\, \mathcal{N}_P \rangle \\ (\mathcal{N} \cup \mathcal{N}_P)(P''.x) = P.P'.x \qquad P''.x \neq P.P'.x \qquad (P.P'.x : T) \in \langle e_1 \dots e_n \rangle \end{array}}{E \,|\, \mathcal{N} \vdash P''.x \rhd_\delta P.P'.x}$$

Bibliography

[1] Davide Ancona and Elena Zucca. A calculus of module systems. *J. Funct. Program.*, 12(2):91–132, 2002.

[2] A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi. A compact kernel for the Calculus of Inductive Constructions. In *Special Issue on Iteractive Proving and Proof Checking of the Academy Journal of Engineering Sciences (Sadhana) of the Indian Academy of Sciences*, pages 71–144. SADHANA (BANGALORE), 2009.

[3] Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. Crafting a Proof Assistant. In Thorsten Altenkirch and Conor McBride, editors, *TYPES*, volume 4502 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 2006.

[4] Clemens Ballarin. Interpretation of locales in isabelle: Managing dependencies between locales. Technical report, Technische Universität München, 2006.

[5] Clemens Ballarin and Technische Universität München. Interpretation of locales in Isabelle: Theories and proof contexts. In *Mathematical Knowledge Management (MKM 2006), LNAI 4108*, pages 31–43. Springer-Verlag, 2006.

[6] S. Berardi. *Type Dependance and Constructive Mathematics*. PhD thesis, Mathematical institute of Torino, 1988.

[7] Stefan Berghofer and Tobias Nipkow. Proof Terms for Simply Typed Higher Order Logic. In Mark Aagaard and John Harrison, editors, *TPHOLs*, volume 1869 of *Lecture Notes in Computer Science*, pages 38–52. Springer, 2000.

[8] Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors. *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*. Springer, 2009.

[9] Ana Bove, Peter Dybjer, and Ulf Norell. A Brief Overview of Agda - A Functional Language with Dependent Types. In Berghofer et al. [8], pages 73–78.

[10] Luca Cardelli, James E. Donahue, Lucille Glassman, Mick J. Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. *SIGPLAN Notices*, 27(8):15–42, 1992.

[11] Jacek Chrząszcz. Implementing Modules in the Coq System. In David A. Basin and Burkhart Wolff, editors, *TPHOLs*, volume 2758 of *Lecture Notes in Computer Science*, pages 270–286. Springer, 2003.

[12] Jacek Chrząszcz. *Modules in Type Theory with Generative Definitions.* PhD thesis, Université Paris-Sud, 2004.

[13] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science.* Springer, 1990.

[14] Judicaël Courant. Explicit Universes for the Calculus of Constructions. In Victor Carreño, César Muñoz, and Sofiène Tahar, editors, *TPHOLs*, volume 2410 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2002.

[15] Judicaël Courant. Mc_2 a module calculus for Pure Type Systems. *J. Funct. Program.*, 17(3):287–352, 2007.

[16] Frank DeRemer and Hans Kron. Programming-in-the large versus Programming-in-the-small. In *Proceedings of the international conference on Reliable software*, pages 114–121, New York, NY, USA, 1975. ACM.

[17] The Coq development team. *Coq Reference Manual.* INRIA, 2008.

[18] Derek Dreyer, Karl Crary, and Robert Harper. A Type System for Higher-Order Modules, 2003.

[19] Pollack *et.al.* The LEGO Proof Assistant, 2001.

[20] William M. Farmer, Joshua D. Guttman, F. Javier, and Thayer Fábrega. IMPS: an updated system description. In *Automated Deduction–CADE-13, Lecture Notes in Computer Science*, pages 298–302. SpringerVerlag, 1996.

[21] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. Little Theories. In *Automated Deduction/CADE-11, volume 607 of Lecture Notes in Computer Science*, pages 567–581. Springer-Verlag, 1992.

[22] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging Mathematical Structures. In Berghofer et al. [8], pages 327–342.

[23] Georges Gonthier. The four colour theorem: Engineering of a formal proof. In Deepak Kapur, editor, *ASCM*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007.

[24] Robert Harper and Mark Lillibridge. A Type-Theoretic Approach to Higher-Order Modules with Sharing. In *POPL*, pages 123–137, 1994.

[25] Robert Harper and John C. Mitchell. On the type structure of standard ML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):211–252, 1993.

[26] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-Order Modules and the Phase Distinction. In *POPL*, pages 341–354, 1990.

[27] Hugo Herbelin. Type Inference with algebraic universes in the Calculus of Inductive Constructions.

[28] Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. *ACM Trans. Program. Lang. Syst.*, 27(5):857–881, 2005.

[29] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle Semantics for Concurrent Separation Logic. In Sophia Drossopoulou, editor, *ESOP*, volume 4960 of *Lecture Notes in Computer Science*, pages 353–367. Springer, 2008.

[30] Gérard Huet. Extending the Calculus of Constructions with Type:Type.

[31] Jacek Chrząszcz. Modules in Coq Are and Will Be Correct. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 130–146. Springer, 2003.

[32] Felix Joachimski and Ralph Matthes. Short Proofs of Normalization for the simply typed lambda-calculus, permutative conversions and Gödel's T. *Arch. Math. Log.*, 42(1):59–87, 2003.

[33] Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. Locales - a sectioning concept for isabelle. In Yves Bertot, Gilles Dowek, André Hirschowitz, C. Paulin, and Laurent Théry, editors, *TPHOLs*, volume 1690 of *Lecture Notes in Computer Science*, pages 149–166. Springer, 1999.

[34] Xavier Leroy. Manifest Types, Modules, and Separate Compilation. In *POPL*, pages 109–122, 1994.

[35] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *22nd symposium Principles of Programming Languages*, pages 142–153. ACM Press, 1995.

[36] Xavier Leroy. A modular module system. *J. Funct. Program.*, 10(3):269–303, 2000.

[37] Xavier Leroy. The Compcert verified compiler, software and commented proof, January 2010.

[38] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems.* PhD thesis, Carnegie Mellon University, 1996.

[39] David B. MacQueen. Modules for Standard ML. In *LISP and Functional Programming*, pages 198–207, 1984.

[40] David B. MacQueen. Using Dependent Types to Express Modular Structure. In *POPL*, pages 277–286, 1986.

[41] David B. MacQueen and Mads Tofte. A Semantics for Higher-Order Functors. In Donald Sannella, editor, *ESOP*, volume 788 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1994.

[42] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[43] John C. Mitchell and Gordon D. Plotkin. Abstract Types Have Existential Type. In *POPL*, pages 37–51, 1985.

[44] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System, 1992.

[45] S. Owre and N. Shankar. Theory Interpretations in PVS. Technical Report SRI-CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park, CA, April 2001.

[46] Frank Pfenning and Christine Paulin-Mohring. Inductively Defined Types in the Calculus of Constructions. In Michael G. Main, Austin Melton, Michael W. Mislove, and David A. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 1989.

[47] R. Pollack. Theories in Type Theory. *Types Workshop on Subtyping, Inheritance and Modular Developpment of Proofs*, 1997.

[48] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, Univ. of Edinburgh, 1994.

[49] Claudio V. Russo. *Types for Modules*. PhD thesis, University of Edinburgh, 1998.

[50] Claudio V. Russo. First-class structures for standard ml. In Gert Smolka, editor, *ESOP*, volume 1782 of *Lecture Notes in Computer Science*, pages 336–350. Springer, 2000.

[51] Amokrane Saïbi. Typing Algorithm in Type Theory with Inheritance. In *POPL*, pages 292–301, 1997.

[52] Paula Severi and Erik Poll. Pure Type Systems with Definitions. In Anil Nerode and Yuri Matiyasevich, editors, *LFCS*, volume 813 of *Lecture Notes in Computer Science*, pages 316–328. Springer, 1994.

[53] Elie Soubiran. A unified framework and a transparent name-space for the Coq module system. In *MLPA '09: Proceedings of the 1st Workshop on Modules and Libraries for Proof Assistants*, pages 38–45, New York, NY, USA, 2009. ACM.

[54] Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2008.

[55] S. Tucker Taft and Robert A. Duff, editors. *Ada 95 Reference Manual, Language and Standard Libraries, International Standard ISO/IEC 8652: 1995(E)*, volume 1246 of *Lecture Notes in Computer Science*. Springer, 1997.

[56] A. Tarski A. Mostowski and R. M. Robinson. *Undecidable theories*. North-Holland, 1953.

[57] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle Framework. In *Theorem Proving in Higher Order Logics, TPHOLs 2008, invited paper, LNCS 5170*. Springer, 2008.