Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Université de Paris-Sud

Jacek Chrząszcz

Modules in Type Theory with Generative Definitions

Supervisors:

Professor Jean-Pierre Jouannaud
Professor Paweł Urzyczyn

Warsaw, September 2003

Pracę niniejszą dedykuję
mojej Mamie i jej marzeniom.

Acknowledgments

First of all, I would like to thank my supervisors, Jean-Pierre Jouannaud and Paweł Urzyczyn. My French supervisor, Jean-Pierre Jouannaud, gave me this interesting subject and guided my work through the years. Without his invaluable help in all aspects of scientific life, this thesis would not have been possible.

I am also very grateful to Paweł Urzyczyn, my Polish supervisor, who brought me to the research area of the λ-calculus. He was a great source of encouragement and support from the very beginning of our collaboration.

During my visits in France, I had many occasions to present and discuss my cloudy thoughts about marying rewriting and modules. I would like to particularly thank Christine Paulin, Gilles Dowek, Judicaël Courant and Frédéric Blanqui for their active part in these discussions and their multiple suggestions.

The practical result of my thesis — implementation of modules in Coq — could not have been done without help and many useful explanations of Coq developers: Hugo Herbelin, Jean-Christophe Filliâtre and Bruno Barras. I am also grateful to Pierre Letouzey for his extraction of modules.

The final result of the many years of my work would not have been possible without help of my wife Daria, who made everything she could to make this dream come true. She was the only person who listened to every detail of every proof contained in this dissertation.

I thank the members of research groups: ZLS in Warsaw, Démons in Orsay and Logical in Roquencourt for their encouragements and a motivating work environment.

I gratefully acknowledge the financial support of the French government and Polish KBN grant no 4 T11C 024 22.

Contents

Chapter 1

Introduction

Program correctness has been considered as a major area for research from the early days of computer science. Because computers play now a central role in our daily life, the need for ensuring that programs behave according to their specification has become a major practical concern.

Because searching for a proof of a proposition is undecidable, program correctness was considered a dream until recently. Proving correctness of very large programs is still out of reach of current technologies. However, proving correctness of chips or critical applications like security protocols or telecommunication protocols is now possible. Indeed, tools exist that allow proving correctness of medium size programs.

The most widely used measure to make sure that a computer program is correct is testing. Unfortunately, as noticed by Dijkstra, "Testing can only show the presence of errors, never their absence." The same thing, to a certain extent, can be said about automatic verification tools. If the tool does not find any errors, what does convince us that the program is correct?

In a proof assistant, the user builds a proof with the help of the computer which takes care of the many tedious details as well as of the proof bookkeeping. The proof assistant will convert the user's indications into a formal proof that will then be checked by the system, using the fact that, contrary to proof search, proof checking is decidable. Besides, the formal proof can be used as a correctness certificate and verified by an independent checker, therefore answering the previous question.

Unfortunately, using any of the existing proof assistants is still quite hard. It is necessary to give very precise hints to the system and things that are obvious for humans often require hours of work to be formally stated and proved. Besides, tools for structuring large proofs are not satisfactory, making proof reuse difficult.

The present thesis addresses this question. We describe a module system similar to those used in programming languages, well suited to proof assistants based on type theory. It permits breaking the development of large proofs into smaller pieces that can later be put together by using composition and instantiation.

This module system is implemented in the Coq proof assistant. It fits a type theory extended with generative definitions, such as inductive types and rewriting, therefore anticipating future versions of Coq: while inductive types have been present for over ten years in Coq, rewriting is still an active research topic, the implementation of which has started only recently.

As any extension of the type theory used for representing and checking proofs, the present one must be proved sound. In this thesis, soundness is obtained relatively to the previously used type theory: it is shown that adding our module system is conservative, that is, does not allow proving propositions that could not have been proved before. In other words, it is possible to obtain new proofs of a given proposition, but these new proofs can indeed be recursively transformed into old ones.

The version 7.4 of Coq with modules can be downloaded from the Coq site. Although this new feature is announced as experimental, it is used by the most heavy users of the system. Some important developments taking advantage of modules have already been realized.

1.1 Formal proofs

Proof assistants are interactive programs, which help users write formal mathematics. They implement a certain logical calculus specifying how to formalize mathematical notions, theorems and proofs, and make sure that every user's input is correct with respect to the underlying logic.

Proof assistants provide their users with a set of procedures called tactics which construct pieces of proofs. Tactics can either be general, like trying to apply any theorem that has already been proved, or specific to a given domain, like proving equalities in group theory or proving instances of propositional tautologies.

A major role of a proof assistant is to make sure that none of the tedious details of a proof is forgotten.

Depending on the status of proofs in the implemented logic and the paradigm on which the proof assistant is built, the proof produced using tactics is either immediately accepted as correct or is reverified by a proof-checker independent from the tactics.

In PVS [88, 78], a proof constructed by the tactics is automatically accepted as correct. Writing tactics is straightforward: given a mathematical statement, the user can write a tactic which answers true/false and no other justification is required by the system.

In Isabelle [52, 82], the proof constructed by tactics is also accepted without rechecking, but in order to construct an object that is accepted as a proof, one has no other way but using tacticals built from basic tactics, which boils down to building a complete proof from the basic tactics themselves, which happen to be derivation rules and axioms of the underlying logic.[1]

In Coq [23, 24], a proof assistant based on the so-called Curry-Howard isomorphism, a proof is an object of the underlying logical formalism, and upon completion, is reverified by a type-checker, called the *kernel* of the proof assistant. In this case, the correctness of

[1]Recently a possibility of using decision procedures has been added to Isabelle, which also allows for accepting theorems without any justification.

a proof depends only on the correctness of the kernel. Since the kernel is rather small in practice, easy to read, and can be itself proved correct, this technology is inherently more reliable than any other.

Curry-Howard isomorphism. In 1958 Curry [37] noticed that typing derivations in simply typed lambda calculus correspond to natural deduction proofs in minimal propositional logic (i.e. using only implication). Moreover, in 1965 Tait [95] saw the correspondence between the computation rule of the lambda calculus, and cut elimination which simplifies the proof obtained by the introduction of an implication followed immediately by its elimination.

More precisely, it turned out that normal proofs (without cuts) 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 example, second-order propositional logic corresponds to Girard's system F of polymorphic types [45] and first order logic can be encoded using dependent types.

This phenomenon is called the *Curry-Howard isomorphism*: it establishes the correspondence between propositions and types, proof and terms, provability and inhabitation, proof normalization and term reduction.

In a proof assistant based on the Curry-Howard isomorphism, such as Coq, 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 The calculus of constructions

In the late 80s, it was noticed that many known type systems for lambda terms differ only in the allowed functional types [7, 96, 4, 5]. A unified syntax and parameterized typing rules was proposed to cover eight type systems, called together the lambda cube.

The syntactic class of pseudoterms, denoted by the letters e and t to give an intuitive distinction between terms and types, is defined as follows:

$$e, t ::= v \mid (e_1\, e_2) \mid [v{:}t]\, e \mid (v{:}t_1)\, t_2 \mid \mathrm{s}$$
$$\mathrm{s} ::= \star \mid \Box$$

A pseudoterm is either a variable, an application of two pseudoterms, an abstraction, a product, or a sort denoted by the letter s. Sorts $\star$ and $\Box$ are two constants. We write abstraction and product in Coq ASCII notation $[v{:}t]\, e$ and $(v{:}t_1)\, t_2$, which corresponds to the standard $\lambda v{:}t.e$ and $\Pi v{:}t_1.t_2$, respectively.

The typing rules tell us how to derive types for terms in a given environment. We will divide them into two parts. The first part deals with the environment, which (in this section) is a sequence of pairs made of a variable and a pseudoterm.

$$\underline{\text{Empty}} \qquad \frac{}{\epsilon \vdash \texttt{ok}}$$

$$\underline{\text{Insert}} \quad \frac{E \vdash \texttt{ok} \qquad E \vdash t : \mathrm{s}}{E; v : t \vdash \texttt{ok}} \qquad\qquad \underline{\text{Lookup}} \quad \frac{E_1; v : t; E_2 \vdash \texttt{ok}}{E_1; v : t; E_2 \vdash v : t}$$

Intuitively, an environment assigns types to variables, and the judgment $E \vdash \texttt{ok}$ means that E is a correct environment.

The second set of rules deals with typing judgments. Note that the Product rule is parametrized by the set $\mathcal{C}$, such that $(\star, \star) \in \mathcal{C} \subseteq \{(\star, \star), (\Box, \star), (\star, \Box), (\Box, \Box)\}$ and that $\approx$ in the Conversion rule means β-convertibility, i.e. the congruence generated by the beta reduction rule $[v{:}t]e\ e' \longrightarrow_\beta e\{v \mapsto e'\}$.

$$\underline{\text{Axiom}} \quad \frac{E \vdash \texttt{ok}}{E \vdash \star : \Box}$$

$$\underline{\text{Product}} \quad \frac{E \vdash t_1 : \mathrm{s}_1 \quad E; v : t_1 \vdash t_2 : \mathrm{s}_2}{E \vdash (v{:}t_1)\ t_2 : \mathrm{s}_2} \quad \text{where} \quad (\mathrm{s}_1, \mathrm{s}_2) \in \mathcal{C}$$

$$\underline{\text{Abstraction}} \quad \frac{E; v : t_1 \vdash e : t_2 \quad E \vdash (v{:}t_1)\ t_2 : \mathrm{s}}{E \vdash [v{:}t_1]e : (v{:}t_1)\ t_2}$$

$$\underline{\text{Application}} \quad \frac{E \vdash e : (v{:}t_1)\ t_2 \quad E \vdash e' : t_1}{E \vdash e\ e' : t_2\{v \mapsto e'\}}$$

$$\underline{\text{Conversion}} \quad \frac{E \vdash e : t \qquad E \vdash t' : \mathrm{s} \qquad E \vdash t \approx t'}{E \vdash e : t'}$$

The sort $\star$ corresponds to the type of all types (or propositions) and the sort $\Box$ to the type of all kinds, $\star$ being the simplest kind expression. The choice of the set $\mathcal{C}$ determines what products[2] can be formed in the calculus. Since $(\star, \star)$ is always an element of $\mathcal{C}$, the product of two types is always a type. Other pairs in $\mathcal{C}$ yield the following characteristics:

- $(\Box, \star)$ corresponds to polymorphism;
- $(\star, \Box)$ corresponds to dependent types;
- $(\Box, \Box)$ corresponds to type constructors.

The Coq proof assistant is based on Coquand and Huet's calculus of constructions [29], the most complex typed lambda calculus of the lambda cube.[3] It has the three above possibilities, making it a formalism capable of encoding higher-order intuitionistic predicate logic. Lambda terms typable in the calculus of constructions can represent all functions provably total in higher-order arithmetics, a property shared with another calculus of the lambda cube due to Girard, called F^ω [45], which does not have dependent types.

[2]Note that product means here a dependent function space rather that a Cartesian product.

[3]The presentation of typing rules with Insert and Lookup follows [85, 28, 29] rather then the definition from [5], where Var and Weak rules are used.

Examples. We will now give a couple of examples illustrating various features of the calculus of constructions. In the examples, we will use the syntax $A \to B$ to denote a product $(v{:}A)\, B$, where v does not occur in B.

According to the Curry-Howard isomorphism, logical propositional variables correspond to the variables of sort $\star$ in the calculus of constructions. Given a variable A, one may prove that $A \to A$ is a correct proposition, i.e:

$$A : \star \vdash A \to A : \star$$

In order to do this, one uses the Product rule for sorts $(\star, \star)$. Moreover, it is possible to prove this proposition:

$$A : \star \vdash [x{:}A]\, x : A \to A$$

Using the Product rule for sorts $(\Box, \star)$, corresponding to polymorphism, one can prove that $(A{:}\star)A$ is a correct proposition. It corresponds to the false proposition, unprovable in the calculus of constructions, i.e. there is no term t such that $\vdash t : (A{:}\star)A$. On the other hand $(A{:}\star)A$ really corresponds to a false proposition, as in any environment E and any proposition P (i.e. a term such that $E \vdash P : \star$), we can prove that $(A{:}\star)A$ implies P. Indeed:

$$E \vdash [f : (A{:}\star)A](f\ P) : ((A{:}\star)A) \to P$$

Using polymorphism one can also encode logical connectives such as conjunction or disjunction. For example, given two propositional variables A and B, the term $(C{:}\star)(A \to B \to C) \to C$ corresponds to the proposition $A \wedge B$: it is possible to construct a proof of $A \wedge B$ from a proof of A and a proof of B and vice-versa.

This encoding allows to form the conjunction of two propositions, but it turns out that polymorphism is not enough to construct the conjunction operator itself. Indeed, if **and** is supposed to be a term such that $E \vdash (\mathbf{and}\ P\ Q) : \star$, then $E \vdash \mathbf{and} : \star \to \star \to \star$. But the latter is only typable (by a sort $\Box$) if the Product rule $(\Box, \Box)$ is allowed. In the calculus of constructions the term **and** is simply

$$[A : \star][B : \star](C{:}\star)(A \to B \to C) \to C$$

The Product rule for $(\star, \Box)$ is required to form predicates. Given a type A (i.e. a term A of type $\star$), every term P of type $A \to \star$ corresponds to a unary predicate over A. A binary predicate of Leibniz equality on the type A can be obtained as the following term of type $A \to A \to \star$:

$$[a : A][b : A](P : A \to \star)(P\ a) \to (P\ b)$$

It becomes also possible to encode datatypes, such as natural numbers, booleans and lists, using the so called *impredicative encoding*. For example, the type of natural numbers is encoded as follows:

$$\begin{aligned} NAT &\equiv (C{:}\star)C \to (C \to C) \to C \\ 0 &\equiv [C{:}\star][z{:}C][s{:}C \to C]z \\ succ &\equiv [n : NAT][C{:}\star][z{:}C][s{:}C \to C](s\ (n\ C\ z\ s)) \end{aligned}$$

Using this encoding, one can define recursive functions, like addition:

$$plus \quad \equiv \quad [n : NAT][m : NAT]\,(n\ NAT\ m\ succ)$$

using the numbers themselves as polymorphic recursors.

Beyond the calculus of constructions. The concept of the λ-cube has been further generalized to the so-called *pure type systems*, or PTS for short [44]. The typing system presented for the λ-cube is parameterized not only by the set of acceptable products, but by a triple consisting of a set of sorts $\mathcal{S}$, a binary relation $\mathcal{A}$ and a ternary relation $\mathcal{R}$ over sorts. The latter two govern the modified typing rules Axiom and Product:

$$\underline{\text{Axiom}} \qquad \frac{E \vdash \mathtt{ok}}{E \vdash \mathrm{s}_1 : \mathrm{s}_2} \text{ if } (\mathrm{s}_1, \mathrm{s}_2) \in \mathcal{A}$$

$$\underline{\text{Product}} \qquad \frac{E \vdash t_1 : \mathrm{s}_1 \quad E; v : t_1 \vdash t_2 : \mathrm{s}_2}{E \vdash (v{:}t_1)\, t_2 : \mathrm{s}_3} \text{ if } (\mathrm{s}_1, \mathrm{s}_2, \mathrm{s}_3) \in \mathcal{R}$$

All other typing rules are unchanged.

Many interesting logical systems can be presented using the pure type system formalism. These include several systems from the AUTOMATH family [97, 38], higher-order logic [20] and many others. The type system implemented in Coq is based on a PTS, where the sort $\star$ is split into two: $\star_P$ and $\star_S$ (which are denoted in Coq as `Prop` and `Set` respectively) to separate the universe of propositions from the universe of types. On the other hand, there is a whole hierarchy of $\Box_i$ for all $i \in \mathbb{N}$ called the *predicative universe hierarchy*. The axioms are $\mathcal{A} = \{\star_P : \Box_i,\ \star_S : \Box_i,\ \Box_i : \Box_j \text{ for any } i < j\}$, and product rules $\{(\star_a, \star_b, \star_b), (\Box_i, \star_b, \star_b), (\star_a, \Box_i, \Box_i), (\Box_i, \Box_j, \Box_k) \text{ for any } \star_a, \star_b = \star_P, \star_S,\ i, j \leq k\}$[4].

Definitions. As we can observe from the examples above, formalizing mathematical notions in a readable form requires the possibility to abbreviate terms by names. This facility was introduced in Coq in the very beginning (already in its precursor called CoC) in the form of global definitions formalized by extending the capabilities of the environment: besides the usual entries of the form $v : t$ assigning types to variables, new entries of the form $v : t := e$ assign a type t and a value e to names of constants. The corresponding typing rules are the following:

$$\underline{\text{Insert}} \ \frac{E \vdash \mathtt{ok} \qquad E \vdash e : t}{E; v : t := e \vdash \mathtt{ok}} \qquad\qquad \underline{\text{Lookup}} \ \frac{E_1; v : t := e; E_2 \vdash \mathtt{ok}}{E_1; v : t := e; E_2 \vdash v : t}$$

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

$$E_1;\ v : t := e;\ E_2 \vdash v \longrightarrow_\delta e$$

[4]In fact the type theory of Coq contains also a cumulativity rule allowing to derive $t : \Box_j$ from $t : \Box_i$ for any $j > i$, which does not match the PTS formalism [26, 64].

and consequently the relation $\approx$ used in the conversion rule becomes $\beta\delta$-equality. Pure type systems with definitions have been formally studied in [93].[5]

Unlike regular variables which be can abstracted, definitions are global and cannot be removed once they have been introduced in the environment. Consequently, a variable declaration of the form $v : t$ which appears in the environment before some definitions, cannot be removed either. One can say that such a variable becomes a global axiom or a global parameter.

Coq concrete syntax for definitions, axioms and parameters is the following:[6]

```
Definition Id : (T:Set)(T→T) := [T:Set][x:T]x.
Axiom a : A.
Parameter A : Prop.
```

1.3 Inductive types

Impredicative encodings as they have been presented in the previous section have many drawbacks both on the usability and logical side: it is quite counter-intuitive to write functions on encoded datatypes; some easy functions cannot be written efficiently, for example the predecessor function on natural numbers performs in linear instead of constant time; it is not possible to prove that $0 \neq 1$; and the induction principle itself is not provable.

Anonymous inductive definitions. For all these reasons primitive inductive types have been added to Coq in version 5.6. Inductive types are packages containing names and types of (mutually defined) inductive types and their constructors. This is denoted by two environments $E^I = v_1^I : t_1 \ldots v_k^I : t_k$ (for inductive types) and $E^C = v_1^c : \hat{t}_1 \ldots v_n^c : \hat{t}_n$ (for constructors), and the whole inductive package is written as $\mathsf{Ind}(E^I := E^C)$.

The first typing rule defines when a package representing an inductive definition is correct.

$$\underline{\text{Formation}} \qquad \frac{E \vdash t_j : \mathrm{s}_j \text{ for } j = 1 \ldots k \qquad E; E^I \vdash \hat{t}_i : \mathrm{s}'_i \text{ for } i = 1 \ldots n}{E \vdash \mathit{Acceptable}(\mathsf{Ind}(E^I := E^C))} \text{ if } \mathsf{POS}_E(E^I := E^C)$$

The side condition $\mathsf{POS}_E(E^I := E^C)$, called *positivity condition*, is a syntactic condition on the kinds of inductive types and types of constructors. Informally, the kind of an inductive type must be a product whose co-domain is a sort, and the type of a constructor must be a product, whose co-domain is one of the inductive types, and whose domain can contain only positive occurrences of the inductive types. Precise formulation of these conditions differs slightly in various papers [80, 81, 100, 24] and since we are not interested in a very precise analysis of inductive types, we do not quote any of them.

[5] In fact the authors studied not only global but also local definitions.

[6] The type of the definition can also be synthesized automatically.

A correct inductive package allows to introduce new kinds of terms of the form $\mathsf{Ind}(E^I := E^C).i$ which are used to represent the inductive types themselves (for $i \leq k$) and their constructors (for $i > k$). Here are the corresponding typing rules:

$$\underline{\text{Inductive Type}} \qquad \frac{E \vdash \mathit{Acceptable}(\mathsf{Ind}(E^I := E^C))}{E \vdash \mathsf{Ind}(E^I := E^C).i : t_i} \text{ if } \ i \leq k$$

$$\underline{\text{Constructor}} \qquad \frac{E \vdash \mathit{Acceptable}(\mathsf{Ind}(E^I := E^C))}{E \vdash \mathsf{Ind}(E^I := E^C).i : \hat{t}_{i-k}\{v_j^I \mapsto \mathsf{Ind}(E^I := E^C).j\}_{j=1\ldots k}} \text{ if } \ i > k$$

Note the substitution in the conclusion of the second rule: the type $\hat{t}_{i-k}$ may contain free occurrences of $v_1^I \ldots v_k^I$, which were previously bound in $\mathsf{Ind}(E^I := E^C)$.

One can now define natural numbers in Peano style, with $E^I = nat : \star$ and $E^C = O : nat;\ S : nat \to nat$. This definition satisfies the positivity condition, and therefore $E \vdash \mathit{Acceptable}(\mathsf{Ind}(E^I := E^C))$.

In order to conveniently use natural numbers, we need to give names to the type itself and its constructors, using global definitions as described in the previous section:

$$\begin{array}{rcl} nat & : & \star \ := \ \mathsf{Ind}(E^I := E^C).1 \\ O & : & nat \ := \ \mathsf{Ind}(E^I := E^C).2 \\ S & : & nat \to nat \ := \ \mathsf{Ind}(E^I := E^C).3 \end{array}$$

Apart from the typing rules for inductive types and their constructors (the latter are also called introduction rules), there are also elimination rules for inductive types. For every $j = 1 \ldots k$ the term $\mathsf{Elim}(\mathsf{Ind}(E^I := E^C), j)$ denotes the eliminator for the j-th inductive type defined in the inductive package. These terms correspond to recursors from the computation point of view and to induction principles from the logical point of view. The corresponding reduction rules are called the ι-rules. Unfortunately, the typing schemes for eliminators are very complicated. Since we will not need them in the rest of the thesis (eliminators can be directly defined by rewriting), we will omit the details here. Instead, we simply give the dependent recursor for the inductive type of natural numbers given above:

$$\begin{array}{l} \mathsf{Elim}(\mathsf{Ind}(E^I := E^C), 1) \\ \qquad\qquad : \ (P : nat \to \star)(P\ O) \to ((n : nat)(P\ n) \to (P(S\ n))) \to (n : nat)(P\ n) \end{array}$$

Its associated reduction rules, after abbreviating $\mathsf{Elim}(\mathsf{Ind}(E^I := E^C), 1)$ by $\mathrm{nat}_{\mathrm{rec}}$, are the following:

$$\begin{array}{rcl} \mathrm{nat}_{\mathrm{rec}}\ P\ f_z\ f_s\ O & \longrightarrow_\iota & f_z \\ \mathrm{nat}_{\mathrm{rec}}\ P\ f_z\ f_s\ (S\ n) & \longrightarrow_\iota & f_s\ (\mathrm{nat}_{\mathrm{rec}}\ P\ f_z\ f_s\ n) \end{array}$$

Named inductive definitions. The inductive definitions presented so far are anonymous. Creating an inductive definition and giving names to its components come separately. This approach follows the λ-calculus tradition of describing functions without giving them names in the first place, and this approach is indeed well suited to a theoretical analysis of inductive types.

From a practical point of view however, anonymous inductive types are very cumbersome since even simple terms containing explicit constructors are very big and therefore unreadable. Until version 6.2 of Coq, abbreviations of anonymous inductive types were not unfolded unless really necessary, but this made the implementation quite complicated and did not always work.

The implementation of inductive definitions in Coq was changed in version 6.2. An inductive definition written by the user became a declaration of names corresponding to the inductive type and its constructors, and the information binding them together in an inductive package appeared in the environment instead of in the terms themselves. This corresponds to the following typing rules:

$$\underline{\text{Insertion}} \qquad \frac{\begin{array}{c} E \vdash \mathtt{ok} \\ E \vdash t_j : \mathrm{s}_j \text{ for } j = 1 \dots k \\ E; E^I \vdash \hat{t}_i : \mathrm{s}'_i \text{ for } i = 1 \dots n \end{array}}{E; \mathsf{IndDef}(E^I := E^C) \vdash \mathtt{ok}} \quad \text{if } \mathsf{POS}_E(E^I := E^C)$$

$$\underline{\text{Inductive Type}} \qquad \frac{E_1; \mathsf{IndDef}(E^I := E^C); E_2 \vdash \mathtt{ok}}{E_1; \mathsf{IndDef}(E^I := E^C); E_2 \vdash v^I_j : t_j}$$

$$\underline{\text{Constructor}} \qquad \frac{E_1; \mathsf{IndDef}(E^I := E^C); E_2 \vdash \mathtt{ok}}{E_1; \mathsf{IndDef}(E^I := E^C); E_2 \vdash v^c_i : \hat{t}_i}$$

Note that there is no more need for a substitution in the Constructor rule: the inductive type names occurring in $\hat{t}_i$ are now valid in the global environment.

Another difference is that named inductive definitions are *generative*: in an environment containing the inductive definition of natural numbers

$$\mathsf{IndDef}(\ nat : \star \ \ := \ \ O : nat;\ S : nat \to nat\ )$$

it is not possible to give a definition of type *nat* without mentioning the name *nat*. In other words, given an environment E, there is no term t typable in E such that $E; \mathsf{IndDef}(E^I := E^C) \vdash t \approx nat$, where $\mathsf{IndDef}(E^I := E^C)$ is the definition of natural numbers.

The concrete Coq syntax for inductive types is very similar to the formal one. Here is the Coq definition of natural numbers:

```
Inductive nat : Set  :=   O : nat   |  S : nat → nat.
```

Another, quite independent change in the treatment of inductive definitions was the split of the elimination constant into two constructions `Cases` and `Fix` implementing respectively case analysis and the fixpoint operator, making the writing of primitive recursive functions much more intuitive. For example the function plus can now be given in the ML-style as:

```
Fix plus/1 : nat→nat→nat :=
   [n:nat][m:nat]Cases n of
                   O ⇒ m
                 | (S n') ⇒ (S (plus n' m))
                 end
```

making the recursion explicit. Of course not every expression is allowed as a body of `Fix`, as this would lead to non-terminating computations. The user must point out a recursion argument that *structurally decreases* in every recursive call. In the case of `plus`, the recursion is made on the first argument (which is pointed out with `/1`) and indeed, as `n` is matched against `(S n')`, `n'` is structurally smaller in the recursive call `(plus n' m)`.

Primitive inductive types together with the possibility of defining types (and propositions) by case analysis on inductive types (called *strong elimination*) overcome the main disadvantages of the impredicative encoding. It is now possible to prove that $0 \neq 1$, induction schemes can be proved using elimination, and writing simple primitive recursive functions follows the ML-style.

1.4 Rewriting

Equality is ubiquitous in mathematics. Yet it turns out that proof assistants based on the Curry-Howard isomorphism are not very good at handling equality. While proving an equality is not a problem in itself, using already established equalities is quite problematic. Apart from equalities resulting from internal reductions (beta, iota), which can be used without being recorded in the proof term by using the Conversion rule, any other use of an equality requires giving all details of its use explicitly in the proof term: the name (proof) of the equality, the place where the equality is used in the form of an abstracted context, and both the left-hand and right-hand sides of the equality. A single equality step may generate a subexpression of the proof term, whose size is proportional to the size of the goal. As a result, proof terms may become extremely large, requiring sometimes more storage than available and making type-checking time consuming: working with equations in Coq is not very convenient.

A straightforward idea for reducing the size of proof terms is to allow using other equalities in the conversion rule, making their use transparent. As usual, this is done in practice by using the user-defined equalities as one-way *rewrite rules*. In order to keep a decidable conversion rule, these rewrite rules must of course be checked for strong normalization and confluence.

Existing techniques for proving strong normalization of rewriting in the calculus of construction [3, 13, 11, 12, 98, 99], following numerous works about rewriting in the simply-typed lambda calculus [53, 54, 55, 14], assume that the rewriting system is given as a whole from the beginning. It is of course very undesirable to impose such a condition on the users of an interactive proof assistant. Fortunately, these termination criteria are modular, meaning that successive rewriting systems may in fact be added one by one. Introducing definitions by rewriting becomes then very similar to giving named inductive types. It requires an environment $E^\Gamma = v_1 : t_1 \ldots v_n : t_n$, containing the names and types of the symbols defined by the rewrite rules, and a set of rules $R = \{E_i \mid e_i^L \longrightarrow e_i^R : \hat{t}_i\}_{i=1\ldots m}$, where $E_i = v_1^i : t_1^i \ldots v_{n_i}^i : t_{n_i}^i$ is a rule environment, e_i^L and e_i^R are the left-hand and right-hand sides of the rule and $\hat{t}_i$ the type common to both sides:

Formation

$$\frac{\begin{array}{c} E \vdash \mathtt{ok} \\ E \vdash t_k : \mathrm{s}_k \quad \text{for } k = 1 \ldots n \\ \left.\begin{array}{cc} E; E_i \vdash \mathtt{ok} & E; E_i \vdash \hat{t}_i : \hat{\mathrm{s}}_i \\ E; E^\Gamma; E_i \vdash e_i^L : \hat{t}_i & E; E^\Gamma; E_i \vdash e_i^R : \hat{t}_i \end{array}\right\} \text{for } i = 1 \ldots m \end{array}}{E; \mathsf{RewDef}(E^\Gamma, R) \vdash \mathtt{ok}} \quad \text{if } \mathsf{ACC}_E(E^\Gamma, R)$$

where $\mathsf{ACC}_E(E^\Gamma, R)$ is an acceptance condition which plays a role similar to the positivity condition for inductive types, by syntactically ensuring strong normalization and confluence of rewriting. Research for coming up with flexible and easy to implement syntactic conditions is still quite active [99, 12]. We will not describe these results here, but rely on the assumption that reasonable criteria exist for that purpose.

Once a definition by rewriting is in the environment, we can use it either to extract the type of a defined function or to perform computations.

$$\underline{\text{Function}} \qquad \frac{E'; \mathsf{RewDef}(E^\Gamma, R); E'' \vdash \mathtt{ok}}{E'; \mathsf{RewDef}(E^\Gamma, R); E'' \vdash v_j : t_j}$$

$$\underline{\text{Computation}} \qquad \frac{E \vdash \mathtt{ok} \qquad \delta : E_i \to E}{E \vdash e_i^L\delta \longrightarrow_R e_i^R\delta}$$

where $E = E'; \mathsf{RewDef}(E^\Gamma, R); E''$ for some E' and E''. The notation $\delta : E_i \to E$ means that $E \vdash v\delta : t\delta$ for every $v : t \in E_i$.

The new reduction relation $\longrightarrow_R$ becomes now part of the conversion relation $\approx$. Using rewriting, it becomes possible to define addition for natural numbers as follows:

```
Symbol plus : nat → nat → nat
Rules
  plus O y ⇒ y
  plus (S x) y ⇒ S (plus x y)
  plus x O ⇒ x
  plus x (S y) ⇒ S (plus x y)
  plus x (plus y z) ⇒ plus (plus x y) z
```

In the above definition the first two rules correspond to the definition by `Cases` and `Fix` presented in the previous section, and the other three represent equalities which are only provable by induction. Therefore using the definition of `plus` based on rewriting becomes much easier than using the one with `Fix` and `Cases`, since all rewrite rules can now be applied automatically in the conversion.

The above syntax is inspired by the experimental "Rewriting" branch of Coq, developed by Fréderic Blanqui. This new facility is not yet provided in the distributed version of Coq however.

1.5 Modules

In order to certify programs, it is important that a proof assistant provides users with facilities to manage large developments. Coq supports multi-file developments and the structure of a file can be nicely presented using sections. Moreover, sections provide a mechanism for abstracting individual definitions and theorems. However, all versions of Coq prior to version 7.4 are lacking the possibility to conveniently define parameterized theories or parametrized certified data structures, which are needed to make development in the large possible in practice.

In functional programming languages from the ML family, such facilities are provided by the module system [65]. Modules group together related definitions, and higher-order modules (functors) make massive abstraction very convenient.

Modules make a small typed language of their own, where module types play a role of interfaces for modules. On one hand they are used to distinguish the definitions of general interest from auxiliary definitions, on the other hand they specify parameters of higher-order modules.

1.5.1 Informal presentation

In this section, we briefly describe the main elements of an ML-style module system, using very simple examples in a Coq-like syntax. We start with basic module expressions, called structures.

Structures. A *structure* corresponds to a list of definitions and lemmas (together with their proofs) enclosed in the keywords `Struct` and `End`. For example, a structure containing a definition of a type `T` and an element `x` of that type is declared as follows:

```
Struct
  Definition T := nat.
  Definition x := O.
End.
```

In the sequel we refer to this module expression as m.

Signatures. A type of a structure is a *signature*. A possible type for the structure m given above could be:

```
Sig
  Definition T := nat.
  Parameter x : T.
End.
```

Note that this signature hides the value of `x`, while showing the value of `T`.

Functors. A functor is a module abstraction. The following functor (further referred to as F) takes as argument a module `X` whose signature contains a type declaration and an element of that type, and returns a function type together with a constant function.

```
Functor [ X : Sig
                Parameter T : Set.
                Parameter x : T.
              End ]
      Struct
        Definition T := X.T → X.T.
        Definition x := [a:X.T] X.x.
      End
```

The notation `X.T` used above denotes the selection of the `T` component of the module `X`. This allows to distinguish between the type `T` of the argument module `X` and the type `T` returned by the functor F.

Functor types. Functors must of course themselves have types. The above functor may be assigned the following functor type:

```
Funsig ( X : SIG ) SIG
```

where *SIG* is the signature

```
 Sig
   Parameter T : Set.
   Parameter x : T.
 End
```

Higher-order functors. The following functor, further referred to as H, is a higher-order functor, applying its first argument, which is itself a functor, to its second argument:

```
Functor [ G : Funsig ( X : SIG ) SIG ]
  Functor [ Y : SIG ]
    (G Y)
```

A higher order module type of this functor could be

```
Funsig ( G : Funsig ( X : SIG ) SIG )
  Funsig ( Y : SIG )
    SIG
```

Subtyping. It is now possible to apply the higher order functor H to the functor F and the module m, forming a correct module expression $(H\ F\ m)$. This application is made possible by the module subtyping mechanism, which allows the module expression m to be typed with the signature *SIG* and consequently to the module expression $(H\ F)$ of type `Funsig(Y:`*SIG*`)`*SIG* to be applied to m. The subtyping relation is written <:

At the signature level, the module subtyping relation allows a signature defining more elements or in a more precise manner to be a subtype of a signature with less elements or a less precise description. For example:

```
Sig
  Definition T:Set:=nat.      <:     Sig
  Parameter x:T.                       Parameter T:Set.
End                                  End
```

because there are more elements on the left than on the right, and the element `T` is more precisely described on the left — as a type equal to `nat` — than on the right. From the second of the above reasons it follows that the left-hand side signature is also a subtype of the signature *SIG*.

At the functor type level, the subtyping relation is a bit more complex, since it is contravariant in the argument type and co-variant in the result type.

In general, one can say that a module system is quite similar to a dependently typed lambda calculus with nested records, record types and monotone record type subtyping. Structures correspond to dependent records, signature to record types, functors to dependent functions and functor types to dependent products.

1.5.2 Module system for type theory

The first module system designed for a proof assistant based on the Curry-Howard isomorphism is due to Judicaël Courant [31, 32, 33]. In this system, modules are anonymous second-class objects: module expressions cannot be abstracted with respect to terms and vice versa. The associated reduction rules enjoy the necessary metatheoretical properties: subject reduction, strong normalization and confluence. As a result, the system obtained by extending an arbitrary pure type system (without inductive types or rewriting) by Courant's module system is conservative. The module system described in this thesis and implemented in Coq is based on Courant's, and we will therefore present this system in detail.

Specifications. In the following we will use the small letter m to denote module expressions (structures and functors) and the capital letter M to denote module types (signatures and functor types).

We already noticed in Section 1.2 that entries in the environment of a proof assistant can have the form of a definition $v : t := e$ or of a variable declaration $v : t$. Given that the same two options are possible for modules and that signatures and structures contain entries similar to those in the environment, the type system would contain many rules having a very similar appearance. In order to limit the complexity of the type system, Courant decided to introduce *specifications*, which are syntactic objects attached to variables in an environment (and signatures and structures) giving whatever the information is available about these variables. Specifications are also used to formalize signature subtyping in a succinct way.

In Courant's calculus the environment is a list of pairs made of a variable and its specification. The variable declaration $v : t$ from Section 1.2 becomes $v : \mathsf{Ty}(t)$ where $\mathsf{Ty}(t)$ is an *abstract* specification, whose meaning is that the object named v is of type t. The definition $v : t := e$ becomes $v : \mathsf{Eq}(e : t)$ where $\mathsf{Eq}(e : t)$ is a *manifest* specification, whose meaning is that the object named v is equal to e of type t.[7]

The rules for building correct environments are now split into abstract formation rules assuming correct specifications, and concrete rules for building correct abstract or manifest specifications. We start with the rules for building environments, E being as usual our favorite letter for denoting an environment:

$$\underline{\text{Empty}} \qquad \frac{}{\epsilon \vdash \mathtt{ok}}$$

$$\underline{\text{Insert}} \qquad \frac{E \vdash \mathtt{ok} \qquad E \vdash S : \mathtt{spec}}{E; v : S \vdash \mathtt{ok}} \qquad\qquad \underline{\text{Lookup}} \qquad \frac{E_1; v : S; E_2 \vdash \mathtt{ok}}{E_1; v : S; E_2 \vdash v : S}$$

In the rules above, S is any specification and $\mathtt{spec}$ is a new sort used in judgments $E \vdash S : \mathtt{spec}$ denoting that S is a correct specification in the environment E.[8]

In the following rule schemes for building correct specifications, we use Greek letters ϕ, Ψ and Ξ. They stand for terms e, types t and PTS sorts s respectively, if we are interested in term specifications, and for modules m, module types M and a sort $\mathtt{modtype}$ if we are interested in module specifications. The new sort $\mathtt{modtype}$ is used in judgments of the form $E \vdash M : \mathtt{modtype}$, whose meaning is that M is a correct module type. The rules are split into two sets, for abstract specifications first:

$$\underline{\text{Formation}} \qquad \frac{E \vdash \Psi : \Xi}{E \vdash \mathsf{Ty}(\Psi) : \mathtt{spec}}$$

$$\underline{\text{Elimination}} \qquad \frac{E \vdash \phi : \mathsf{Ty}(\Psi)}{E \vdash \phi : \Psi}$$

$$\underline{\text{Satisfaction}} \qquad \frac{E \vdash \phi : \Psi \qquad E \vdash \mathsf{Ty}(\Psi) : \mathtt{spec}}{E \vdash \phi : \mathsf{Ty}(\Psi)}$$

and then for manifest specifications:

$$\underline{\text{Formation}} \qquad \frac{E \vdash \phi : \Psi}{E \vdash \mathsf{Eq}(\phi : \Psi) : \mathtt{spec}}$$

[7]For subjective esthetic reasons, we changed a bit the notation adopted by Courant. In the original, the specification $\mathsf{Eq}(e : t)$ was written $\mathsf{Ty}(t)\mathsf{Eq}(e)$.

[8]Our `spec` replaces `wfs` from [32], which stands for well-formed specification.

$$\underline{\text{Elimination}} \quad \frac{E \vdash \phi' : \mathsf{Eq}(\phi : \Psi)}{E \vdash \phi' : \Psi}$$

$$\underline{\text{Satisfaction}} \quad \frac{E \vdash \phi' \approx \phi \qquad E \vdash \phi' : \Psi \qquad E \vdash \mathsf{Eq}(\phi : \Psi) : \mathtt{spec}}{E \vdash \phi' : \mathsf{Eq}(\phi : \Psi)}$$

$$\underline{\text{Computation}} \quad \frac{E \vdash \phi' : \mathsf{Eq}(\phi : \Psi)}{E \vdash \phi' \approx \phi}$$

Note that an abstract/manifest specification Formation followed by an Insert corresponds to the Insert rules for variables/constants presented in Section 1.2. Similarly, a Lookup followed by an Elimination corresponds to the Lookup rule from Section 1.2. And a Lookup followed by the Computation rule for manifest specifications allows to derive the equality corresponding to the δ-reduction from Section 1.2. Finally, the Satisfaction rules are necessary for forming structures and for signature subtyping as we will see in the next paragraphs.

Structures and signatures. Structures, as they have been introduced in the informal presentation, contain definitions and lemmas with their proofs, and possibly sub-modules. In other words they correspond to *closed* developments. Note that definitions and lemmas are quite different in essence. In the case of definitions, the equality between the name of the constant and its body is important and may be used for typing terms; in the case of lemmas, the only important thing is that the lemma has a proof: the equality between the name of the lemma and its proof is never going to be used for typing.

Therefore, a component of a structure consists of a name, its specification and its implementation. The specification provides the information available for typing subsequent components of the structure, and the implementation provides a witness or a proof of validity of the specification. A lemma v stating the property t with its proof e will be a structure component of the form $v : \mathsf{Ty}(t) := e$, and a definition v' of type t' and body e' will be a structure component of the form $v' : \mathsf{Eq}(e' : t') := e'$. Although it may seem strange that the body e' is both a part of the specification and the witness, this duplication makes typing rules for structures simple and elegant.

A signature, on the other hand, contains the information relevant for typing only, i.e. the names of its components and their specifications.

Before giving the formal syntax for structures and signatures, we have to note that the concrete syntax given in the examples at the beginning of the present section is not sufficient to express certain nested structures and signatures. For example, in the informal signature

```
Sig
  Parameter T:Set.
  Parameter x:T.
  Module N : Sig
               Parameter T:Set.
               Definition y:=x.
               Parameter z:T.
```

```
                    End.
    End
```

further referred to as ***SIG'***, it is impossible to assign a type to the component `y` of the internal module. Its type should be the `T` defined in the main signature, but there is no way to refer to it after the identifier `T` has been redefined in the submodule. A simple solution would be to rename either of the two `T` components, but this is not always possible, because of the possible interaction with functors. Following [48] and [32], our formal syntax will therefore make a difference between labels visible from the exterior and α-convertible binders used inside a given structure or signature, as follows:

$$\mathsf{Struct}\ \mathtt{v}_1 \triangleright v_1 : S_1 := \phi_1 \ldots \mathtt{v}_n \triangleright v_n : S_n := \phi_n\ \mathsf{End}$$

$$\mathsf{Sig}\ \mathtt{v}_1 \triangleright v_1 : S_1 \ldots \mathtt{v}_n \triangleright v_n : S_n\ \mathsf{End}$$

where $\mathtt{v}$ denotes an external label, v an internal binder, S a specification and ϕ an implementation, which can be a term e or a module expression m. From this point on we assume that unless stated otherwise, all variables v bound in environments, structures, signatures, lambdas and functors are different. In the syntax with labels and binders, the signature ***SIG'*** can be written as follows:

$$\begin{array}{l}
\mathsf{Sig} \\
\quad \mathtt{T} \triangleright T : \mathsf{Ty}(\mathtt{Set}) \\
\quad \mathtt{x} \triangleright x : \mathsf{Ty}(T) \\
\quad \mathtt{N} \triangleright N\ :\ \mathsf{Ty}(\ \mathsf{Sig}\quad \mathtt{T} \triangleright T' : \mathsf{Ty}(\mathtt{Set})\quad \mathtt{y} \triangleright y : \mathsf{Eq}(x : T)\quad \mathtt{z} \triangleright z : \mathsf{Ty}(T')\quad \mathsf{End}\ ) \\
\mathsf{End}
\end{array}$$

making the name clash disappear. Because it is a bit heavy, we do not intend to follow this formal syntax in the examples: we will actually abbreviate $\mathtt{b} \triangleright b$ as `b`.

A signature M is correct, which is asserted by the judgment $E \vdash M : \mathtt{modtype}$, if its labels are disjoint and the component specifications are correct. This yields the following typing rule for signature formation:[9]

$$\underline{\text{Formation}} \qquad \frac{E; v_1 : S_1 \ldots v_n : S_n \vdash \mathtt{ok}}{E \vdash \mathsf{Sig}\ \mathtt{v}_1 \triangleright v_1 : S_1 \ldots \mathtt{v}_n \triangleright v_n : S_n\ \mathsf{End} : \mathtt{modtype}}$$

$$\text{if } \mathtt{v}_1 \ldots \mathtt{v}_n \text{ are pairwise disjoint labels}$$

A structure is correct if so are its component specifications, and if each implementation ϕ_k satisfies its specification S_k, possibly using the information contained in the specifications $S_1 \ldots S_{k-1}$. The yields the following typing rule for structure formation:

[9] In [32] there are two rules: for forming an empty signature and for adding a specification to a signature. We decided to give the more intuitive single-rule presentation with dots. Both approaches are of course equivalent. The same difference in presentation concerns the Structure rule and the definition of δ-reduction.

Structure

$$\frac{\begin{array}{c} E \vdash \mathsf{Sig}\ \mathtt{v}_1 \triangleright v_1 : S_1 \ldots \mathtt{v}_n \triangleright v_n : S_n\ \mathsf{End} : \mathtt{modtype} \\ E; v_1 : S_1 \ldots v_{k-1} : S_{k-1} \vdash \phi_k : S_k \quad \text{for } k = 1 \ldots n \end{array}}{E \vdash \mathsf{Struct}\ \mathtt{v}_1 \triangleright v_1 : S_1 := \phi_1 \ldots \mathtt{v}_n \triangleright v_n : S_n := \phi_n\ \mathsf{End}\ :\ \mathsf{Sig}\ \mathtt{v}_1 \triangleright v_1 : S_1 \ldots \mathtt{v}_n \triangleright v_n : S_n\ \mathsf{End}}$$

Once a module expression m has been proved to satisfy a given signature, we may extract its components:

Access
$$\frac{E \vdash m : \mathsf{Sig}\ \mathtt{v}_1 \triangleright v_1 : S_1 \ldots \mathtt{v}_n \triangleright v_n : S_n\ \mathsf{End}}{E \vdash m.\mathtt{v}_k : S_k\{v_i \mapsto m.\mathtt{v}_i\}_{i=1\ldots k-1}}$$

The substitution in the conclusion is needed since S_k may contain free occurrences of the variables $v_1 \ldots v_{k-1}$. For example, given a module K of signature ***SIG'***, defined as above, it is possible to derive the specification of $K.\mathtt{x}$ to be $\mathsf{Ty}(K.\mathtt{T})$.

Functors. Like lambda abstractions are functional terms, functors are functional modules, and therefore functor types correspond to dependent products. Rules for forming functor types and for typing functors and their applications are very similar to the rules for products, term abstraction and application:

Formation
$$\frac{E \vdash M_1 : \mathtt{modtype} \qquad E; v : \mathsf{Ty}(M_1) \vdash M_2 : \mathtt{modtype}}{E \vdash \mathsf{Funsig}(v{:}M_1)\ M_2 : \mathtt{modtype}}$$

Functor
$$\frac{E; v : \mathsf{Ty}(M_1) \vdash m : M_2 \qquad E \vdash \mathsf{Funsig}(v{:}M_1) M_2 : \mathtt{modtype}}{E \vdash \mathsf{Functor}[v{:}M_1]\, m : \mathsf{Funsig}(v{:}M_1) M_2}$$

Application
$$\frac{E \vdash m : \mathsf{Funsig}(v : M_1)\ M_2 \qquad E \vdash m' : M_1}{E \vdash m\ m' : M_2\{v \mapsto m'\}}$$

Subtyping. Before defining the subtyping for signatures and functor types, let us see its use via the following subsumption rule. It says that a module expression of type M_1 is of any module type M_2 that is a supertype of M_1:

Subsumption
$$\frac{E \vdash m : M_1 \quad E \vdash M_1 <: M_2}{E \vdash m : M_2}$$

For signatures, subtyping corresponds to field extension:

Signatures

$$\frac{\begin{array}{c} E \vdash \mathsf{Sig}\ \mathtt{v}_1 \triangleright v_1 : S_1 \ldots \mathtt{v}_n \triangleright v_n : S_n\ \mathsf{End} : \mathtt{modtype} \\ E \vdash \mathsf{Sig}\ \mathtt{v}'_1 \triangleright v'_1 : S'_1 \ldots \mathtt{v}'_{n'} \triangleright v'_{n'} : S'_{n'}\ \mathsf{End} : \mathtt{modtype} \\ \{\mathtt{v}'_1 \ldots \mathtt{v}'_{n'}\} \subseteq \{\mathtt{v}_1 \ldots \mathtt{v}_n\} \\ \text{by } \alpha\text{-conversion, we assume } \mathtt{v}'_{i'} = \mathtt{v}_i \text{ implies } v'_{i'} = v_i \\ E; v_1 : S_1 \ldots v_n : S_n \vdash v'_k : S'_k \quad \text{for } k = 1 \ldots n' \end{array}}{E \vdash \mathsf{Sig}\ \mathtt{v}_1 \triangleright v_1 : S_1 \ldots \mathtt{v}_n \triangleright v_n : S_n\ \mathsf{End}\ <:\ \mathsf{Sig}\ \mathtt{v}'_1 \triangleright v'_1 : S'_1 \ldots \mathtt{v}'_{n'} \triangleright v'_{n'} : S'_{n'}\ \mathsf{End}}$$

A more precise but less readable version of this rule uses an explicit injection on indices describing the inclusion between the sets of labels $\{\mathtt{v}'_1 \dots \mathtt{v}'_{n'}\}$ and $\{\mathtt{v}_1 \dots \mathtt{v}_n\}$ instead of this informal reference to α-conversion.[10]

Note that the premises $E; v_1 : S_1 \dots v_n : S_n \vdash v'_k : S'_k$ can be derived using the specification satisfaction rule allowing in particular to derive $v_k : \mathsf{Ty}(t)$ when the specification of v_k in the environment is $\mathsf{Eq}(e : t)$. For module components the Subsumbption rule may also be used.

For functor types, subtyping is contravariant in the argument type and covariant in the result type:

$$\underline{\text{Functor types}} \qquad \frac{E \vdash M'_1 <: M_1 \quad E; v : \mathsf{Ty}(M'_1) \vdash M_2 <: M'_2}{E \vdash \mathsf{Funsig}(v{:}M_1)\ M_2 <: \mathsf{Funsig}(v{:}M'_1)\ M'_2}$$

This rule can be understood as follows: given a functor transforming a witness of the module type M_1 into a witness of the module type M_2, the same functor can be seen as transforming a witness of a richer module type M'_1 into a witness of a poorer module type M'_2.

Strengthening. Although the module system given so far looks satisfactory, it will reject the following natural definitions:

$$\begin{array}{lcl} \mathtt{X} & : & \mathsf{Ty}(\ \mathsf{Sig}\ \mathtt{T} : \mathsf{Ty}(\mathtt{Set})\ \mathsf{End}\ ) \ := \ \mathsf{Struct}\ \mathtt{T} : \mathsf{Ty}(\mathtt{Set}) := \mathtt{nat}\ \mathsf{End} \\ \mathtt{Y} & : & \mathsf{Ty}(\ \mathsf{Sig}\ \mathtt{T} : \mathsf{Eq}(X.\mathtt{T} : \mathtt{Set})\ \mathsf{End}\ ) \ := \ X \end{array}$$

Indeed, the typing system given so far does not allow to derive the following obvious statement:

$$X \ : \ \mathsf{Sig}\ \mathtt{T} : \mathsf{Eq}(X.\mathtt{T} : \mathtt{Set})\ \mathsf{End}$$

To remedy this situation, we need the following strengthening rule:

$$\underline{\text{Strengthening}} \qquad \frac{E \vdash m : M}{E \vdash m : M/m}$$

where the notations M/m and S/ϕ are defined together as follows:

$$\begin{array}{rcl} \mathsf{Sig}\ \mathtt{v}_1 \triangleright v_1 : S_1 \dots \mathtt{v}_n \triangleright v_n : S_n\ \mathsf{End}/m & = & \mathsf{Sig}\ \mathtt{v}_1 \triangleright v_1 : S_1/m.\mathtt{v}_1; \dots; \mathtt{v}_n \triangleright v_n : S_n/m.\mathtt{v}_n\ \mathsf{End} \\ (\mathsf{Funsig}(v{:}M_1)\ M_2)/m & = & \mathsf{Funsig}(v{:}M_1)\ (M_2/(m\ v)) \\ \mathsf{Eq}(\phi' : \Psi)/\phi & = & \mathsf{Eq}(\phi' : \Psi/\phi) \\ \mathsf{Ty}(\Psi)/\phi & = & \mathsf{Eq}(\phi : \Psi/\phi) \\ t/e & = & t \end{array}$$

Using Strengthening, one can now show the desired module typing above, since

$$\mathsf{Sig}\ \mathtt{T} : \mathsf{Ty}(\mathtt{Set})\ \mathsf{End}/X \ = \ \mathsf{Sig}\ \mathtt{T} : \mathsf{Ty}(\mathtt{Set})/X.\mathtt{T}\ \mathsf{End} \ = \ \mathsf{Sig}\ \mathtt{T} : \mathsf{Eq}(X.\mathtt{T} : \mathtt{Set})\ \mathsf{End}$$

It can be shown that we can derive $E \vdash M/m <: M$ whenever $E \vdash m : M$ is itself derivable.

[10] The presentation with an explicit injection can be found in [32].

Convertibility. We now come to the reduction rules needed for these anonymous modules. First of all, β-reduction applies to module expressions:[11]

$$\mathsf{Functor}[v : M]m\ m' \longrightarrow_{\beta_m} m\{v \mapsto m'\}$$

The convertibility relation $\approx$ is now defined to be the smallest equivalence relation closed by context, containing β-equality on terms and β_m-equality on modules, as well as the equality following from the Computation rule for $\mathsf{Eq}(\cdot)$ specifications.

In fact, the latter part of the convertibility relation may also be defined as a reduction. The reduction is of course called δ, but due to the complexity of the module calculus, its definition requires 7 rules[12] deriving judgments of the form $E \vdash_\delta m : M$ and $E \vdash_\delta \phi : S$. We give the complete definition below, which will be used in Section 6.3.

Definition 1.5.1. *Let* delta *reduction* $\longrightarrow_\delta$ *be defined as the smallest congruence such that* $E \vdash \phi \longrightarrow_\delta \phi'$ *if* $E \vdash_\delta \phi : \mathsf{Eq}(\phi' : \Psi)$ *is derivable in the following inference system:*

[/S/ACCESS]

$$\frac{E \vdash_\delta m : \mathsf{Sig}\ \mathtt{v}_1 \triangleright v_1 : S_1 \ldots \mathtt{v}_n \triangleright v_n : S_n\ \mathsf{End}}{E \vdash_\delta m.\mathtt{v}_i : S_i\{v_j \mapsto m.\mathtt{v}_j\}_{j=1..i-1}}$$

[/S/VAR]

$$\frac{}{E; v : S; E' \vdash_\delta v : S}$$

[/S/ABS]

$$\frac{E \vdash_\delta m : M}{E \vdash_\delta m : \mathsf{Ty}(M)} \text{ if } m \text{ is not a variable } v \text{ or a selection } m'.\mathtt{v}$$

[/T/SPEC]

$$\frac{E \vdash_\delta m : S \qquad S = \mathsf{Ty}(M) \text{ or } \mathsf{Eq}(m' : M)}{E \vdash_\delta m : M} \text{ if } m \text{ is a variable } v \text{ or a selection } m'.\mathtt{v}$$

[M/T/APP]

$$\frac{E \vdash_\delta m_1 : \mathsf{Funsig}(v : M)M'}{E \vdash_\delta (m_1\ m_2) : M'\{v \mapsto m_2\}}$$

[M/T/LAM]

$$\frac{E; v : \mathsf{Ty}(M) \vdash_\delta m : M'}{E \vdash_\delta \mathsf{Functor}[v : M]m : \mathsf{Funsig}(v : M)M'}$$

[M/T/STRUCT]

$$\frac{}{E \vdash_\delta \mathsf{Struct}\ \mathtt{v}_1 \triangleright v_1 : S_1 := \phi_1 \ldots \mathtt{v}_n \triangleright v_n : S_n := \phi_n\ \mathsf{End}\ :\ \mathsf{Sig}\ \mathtt{v}_1 \triangleright v_1 : S_1 \ldots \mathtt{v}_n \triangleright v_n : S_n\ \mathsf{End}}$$

[11] In [32] it is called μ, but in the later paper [33] its name is changed to β_m.

[12] The rule [M/T/STRUCT] replaces two rules [M/T/EMPTY] and [M/T/DECL] in [32]. See footnote 9.

These rules allow to δ-reduce the term $(F\ m).\mathtt{v}$ to $m.\mathtt{t}$ in the environment containing the declaration

$$F : \mathsf{Ty}(\ \mathsf{Funsig}(X : M)\mathsf{Sig}\ \mathtt{v} : \mathsf{Eq}(X.\mathtt{t} : \mathtt{Set})\ \mathsf{End}\ )$$

On the other hand, it is not possible to derive the statement $E \vdash v \longrightarrow_\delta v$, even though obviously $E \vdash v : \mathsf{Eq}(v : \Psi)$ is derivable for any v in E, using the Satisfaction rule.

Courant shows that δ and β_m reductions together have the subject reduction property, are strongly normalizing, and are confluent modulo β on well-typed expressions. Given an arbitrary normalizing PTS, these properties yield an algorithm for deciding convertibility for the associated module calculus.

Conservativity. The conservativity of the module calculus over the base PTS is shown using yet another reduction relation on modules, called ρ-reduction:

$$m.\mathtt{v}_i \ \longrightarrow_\rho\ \phi_i\{v_j \mapsto m.\mathtt{v}_j\}_{j=1\ldots i-1}$$

where $m = \mathsf{Struct}\ \mathtt{v}_1 \triangleright v_1 : S_1 := \phi_1 \ldots \mathtt{v}_n \triangleright v_n : S_n := \phi_n\ \mathsf{End}$. This reduction is indeed necessary for the proof of conservativity, because otherwise there are irreducible terms of basic type containing module expressions, such as

$$\mathsf{Struct}\ \mathtt{v} : \mathsf{Ty}(t) := e\ \mathsf{End}.\mathtt{v}$$

Courant shows that this reduction relation also has the subject reduction property, and that $\rho\beta_m$ reductions are strongly normalizing. Consequently, given a term of base type, its module subterms may be reduced until they eventually disappear. Therefore, any base type inhabited in the calculus with modules is inhabited in the calculus without modules.

1.6 Anonymous modules, inductive types and rewriting

The calculus of anonymous modules is very appealing from the theoretical point of view: it naturally extends the λ-calculus approach in which defining an entity (e.g., a function) does not necessary mean naming it. Besides, modules expressions are true citizens of type theory rather than a bare practical feature of proof assistants.

On the other hand, the fact that modules are anonymous is hardly useful in practice. Modules are by nature large and complex syntactic objects so terms containing them become quite unreadable. Fortunately, the module specifications make it possible to use modules in such a way, that only simple module expressions appear in terms, a property which is stable by reduction: if every module definition $v : S := m$ is given with an abstract specification, i.e. $S = \mathsf{Ty}(M)$, then no reduction can substitute the body m in a term referring to the module name v.

Consequently, one can implement a module system with some simple syntactic restrictions, like imposing abstract specifications on module components and ruling out complex module expressions from terms, and still benefit from the conservativity proof of Courant.

On the other hand, before implementing a module system for Coq one has to make sure it is compatible not only with the underlying pure type system, but also with inductive types and rewriting which is an important future extension of Coq. In his thesis Courant argues that extending his calculus with anonymous inductive types should be quite straightforward. Inspired by the typing rules for inductive types one could also define anonymous rewriting in a similar way and still expect that the conservativity result on modules would hold.

Anonymous rewrite rules have already been studied in the literature under the name of Rho Calculus [21]. In this framework, a rewrite rule is a binder (like λ in the λ-calculus) which is "used up" when applied. Although this formalism unifies very cleanly the λ-calculus with rewriting, it is not suitable for our purpose. It is for example impossible in the typed version of the Rho Calculus [22] to define a function *plus* such that for all natural numbers n, m the application $plus\ S^n(O)\ S^m(O)$ reduces to $S^{n+m}(O)$ (where $S^i(O)$ means of course S applied i times to O)[13]. Since in our calculus, rewriting aims at subsuming definitions by fixpoint (the `Fix` operator from Coq) we adopted a different approach, in which function symbols have associated rewrite rules that can be applied as many times as necessary.

In this section, we investigate addition of anonymous inductive definitions and anonymous definitions by rewriting to the calculus of modules. We will see that existing specifications are not sufficient to make the extended system usable and we will propose new kinds of specifications to remedy this situation. We will also see that the closure properties imposed by module reductions on acceptance conditions for rewrite rules are far too strong for termination criteria existing in the literature.

Anonymous rewriting. Let us recall the definition of addition on natural numbers given in Section 1.4. Let E^Γ denote the environment $plus : \mathsf{Ty}(nat \to nat)$ and R be the set of rewrite rules:

```
plus O y ⇒ y
plus (S x) y ⇒ S (plus x y)
plus x O ⇒ x
plus x (S y) ⇒ S (plus x y)
plus x (plus y z) ⇒ plus (plus x y) z
```

The definition package, $\mathsf{Rew}(E^\Gamma, R)$, could than be used in a way similar to inductive packages, i.e. $\mathsf{Rew}(E^\Gamma, R).1$ would be a term denoting the function *plus*. This notation $\mathsf{Rew}(E^\Gamma, R).i$ is indeed necessary for handling mutually defined functions.

Rules defined inside a package would yield reductions valid outside the package. For example, the first rule could be used to make the reduction

$$\mathsf{Rew}(E^\Gamma, R).1\ \ O\ \ (S\ (S\ O)) \quad \longrightarrow_R \quad (S\ (S\ O))$$

[13] Such a definition is possible in the untyped Rho Calculus, but it has to contain a non-terminating term, similar to the fixpoint combinator Y of the untyped λ-calculus.

while the last rule could be used to make the more complex reduction

$$\mathsf{Rew}(E^\Gamma, R).1 \;\; x \;\; (\mathsf{Rew}(E^\Gamma, R).1 \;\; y \;\; z) \quad \longrightarrow_R \quad \mathsf{Rew}(E^\Gamma, R).1 \;\; (\mathsf{Rew}(E^\Gamma, R).1 \;\; x \;\; y) \;\; z$$

Old specifications. Unfortunately, unlike modules, anonymous inductive definitions and anonymous definitions by rewriting cannot be used conveniently through specifications. Taking again our favorite example of natural numbers, a module defining them could either be given a completely abstract signature:

$$N' : \mathsf{Ty}(\ \mathsf{Sig}\ \mathtt{nat} : \mathsf{Ty}(\mathtt{Set});\ \mathtt{0} : \mathsf{Ty}(nat);\ \mathtt{S} : \mathsf{Ty}(nat \to nat)\ \mathsf{End}\ )$$

or a completely transparent signature:

$$\begin{array}{l} N'' : \mathsf{Ty}(\ \mathsf{Sig} \\ \qquad\qquad \mathtt{nat} : \mathsf{Eq}(\mathsf{Ind}(E^I := E^C).1 : \mathtt{Set}); \\ \qquad\qquad \mathtt{0} : \mathsf{Eq}(\mathsf{Ind}(E^I := E^C).2 : nat); \\ \qquad\qquad \mathtt{S} : \mathsf{Eq}(\mathsf{Ind}(E^I := E^C).3 : nat \to nat) \\ \qquad\quad \mathsf{End}\ ) \end{array}$$

In the first case the information about the inductive character of natural numbers is completely hidden and there is no way one can have any syntactic constructions for inductive types like `Cases` and `Fix`. In the second case $N''.\mathtt{nat}$, $N''.\mathtt{0}$ and $N''.\mathtt{S}$ simply reduce to their anonymous bodies as described in Section 1.3, yielding terms whose reading is difficult.

The same happens with rewriting. A module containing the definition of *plus* can either be given an abstract signature

$$P' : \mathsf{Ty}(\ \mathsf{Sig}\ \mathtt{plus} : \mathsf{Ty}(nat \to nat)\ \mathsf{End}\ )$$

or a completely transparent one

$$P'' : \mathsf{Ty}(\ \mathsf{Sig}\ \mathtt{plus} : \mathsf{Eq}(\mathsf{Rew}(E^\Gamma, R).1 : nat \to nat)\ \mathsf{End}\ )$$

In the first case $P'.\mathtt{plus}$ cannot be used to calculate and in the second case $P''.\mathtt{plus}$ can be unfolded to the unreadable expression $\mathsf{Rew}(E^\Gamma, R).1$.

New specifications. To remedy this unsatisfactory situation, we need to introduce two new kinds of specifications, $\mathsf{IndSpec}(E^I := E^C)$ and $\mathsf{RewSpec}(E^\Gamma, R)$, with the respective intended meanings "these elements form an inductive definition" and "these elements have rewrite rules R attached to them". Using these new specifications, we could give the following signatures to modules defining natural numbers and addition:

$$\begin{array}{lcl} N''' & : & \mathsf{Ty}(\ \mathsf{Sig}\ \mathtt{nat}, \mathtt{0}, \mathtt{S} : \mathsf{IndSpec}(E^I := E^C)\ \mathsf{End}\ ) \\ P''' & : & \mathsf{Ty}(\ \mathsf{Sig}\ \mathtt{plus} : \mathsf{RewSpec}(E^\Gamma, R)\ \mathsf{End}\ ) \end{array}$$

Consequently, the expressions $N'''.\mathtt{nat}$, $N'''.\mathtt{0}$ and $N'''.\mathtt{S}$ are irreducible, but the fact that they form an inductive definition follows from the form of their specification. Similarly

P'''.plus itself is irreducible, but the reduction rules for plus are available for P'''.plus. For example, using the second rewrite rule from R, the following reduction applies:

$$P'''.\texttt{plus}\ (S\ O)\ O \quad\longrightarrow_R\quad S\ (P'''.\texttt{plus}\ O\ O)$$

Before giving typing rules for $\mathsf{IndSpec}(E^I := E^C)$ and $\mathsf{RewSpec}(E^\Gamma, R)$, let us discuss how these specifications should behave with respect to strengthening.

The result of strengthening $\mathsf{Sig}\ \texttt{plus} : \mathsf{RewSpec}(E^\Gamma, R)\ \mathsf{End}$ by P''' should both indicate that plus is equal to P'''.plus and that plus is defined by rewriting. The solution consists in wrapping $\mathsf{RewSpec}(E^\Gamma, R)$ in either an abstract or a manifest specification. So, in the true formal syntax our modules N''' and P''' should be given the following specifications:

$$\begin{array}{lcl} N''' & : & \mathsf{Ty}(\ \mathsf{Sig}\ \texttt{nat}, \texttt{O}, \texttt{S} \triangleright nat, O, S : \mathsf{Ty}(\mathsf{IndSpec}(E^I := E^C))\ \mathsf{End}\ ) \\ P''' & : & \mathsf{Ty}(\ \mathsf{Sig}\ \texttt{plus} \triangleright plus : \mathsf{Ty}(\mathsf{RewSpec}(E^\Gamma, R))\ \mathsf{End}\ ) \end{array}$$

and the strengthened signatures of N''' and P''' would be respectively

$$\mathsf{Sig}\ \texttt{nat}, \texttt{O}, \texttt{S}\ \triangleright\ nat, O, S\ :\ \mathsf{Eq}(P'''.\texttt{nat}, P'''.\texttt{O}, P'''.\texttt{S}\ :\ \mathsf{IndSpec}(E^I := E^C))\ \mathsf{End}$$
$$\mathsf{Sig}\ \texttt{plus}\ \triangleright\ plus\ :\ \mathsf{Eq}(P'''.\texttt{plus}\ :\ \mathsf{RewSpec}(E^\Gamma, R))\ \mathsf{End}$$

1.6.1 Blocks and their typing

We have seen that term specifications have insufficient expressivity to conveniently use inductive definitions and definitions by rewriting in the calculus of modules. It turns out that one needs a means to specify a whole sequence of terms at the same time.

Besides, expressions like $\mathsf{Ind}(E^I := E^C)$ and $\mathsf{Rew}(E^\Gamma, R)$ represent a sequence of mutually defined terms. Therefore, in order to make the typing system more compact, we decided to create a new syntactic class of *blocks*, representing sequences of terms, possibly mutually defined. A block can be either a *basic block*, i.e. an inductive definition $\mathsf{Ind}(E^I := E^C)$ or a definition by rewriting $\mathsf{Rew}(E^\Gamma, R)$, or a normal sequence of terms.

For blocks, we introduce the same hierarchy that we have for terms and modules. We have terms, types, sorts (which happen to be the same syntactic class), and term specifications $\mathsf{Ty}(t)$ and $\mathsf{Eq}(e : t)$. We have module expressions, module types, the sort modtype, and module specifications $\mathsf{Ty}(M)$ and $\mathsf{Eq}(m : M)$. For blocks (denoted by B), we introduce block types $\mathsf{IndSpec}()$ and $\mathsf{RewSpec}()$ (denoted by BT), a new sort blocktype and block specifications $\mathsf{Ty}(BT)$ and $\mathsf{Eq}(B : BT)$.

As we have already mentioned, we can build a block from individual terms: a block being a sequence of terms will be denoted by $\overline{e}$. We can also extract individual terms from a block B, using the selection operation, denoted by $B.i$. If B is a basic block, we get terms $\mathsf{Ind}(E^I := E^C).i$ or $\mathsf{Rew}(E^\Gamma, R).i$ and if B is a sequence of terms, $\overline{e}.i$ is the i^{th} term of the sequence.

In order to present the calculus with these new specifications, we modify slightly the syntax of environments, signatures and structures. An environment is now a sequence of pairs: a sequence of variables $\overline{v}$ and a specification S. For term and module environment entries the sequences $\overline{v}$ will be of length 1. Similarly, signatures assign a specification to a sequence of labels $\overline{\mathtt{v}}$ and a sequence of internal binders $\overline{v}$ (of equal length). Structures, apart from specifications, assign also an implementation, which is now a term, a module expression, or a block.

We are now going to present the typing rules for blocks. For inductive blocks they are very similar to those for inductive definitions presented in Section 1.3. Let $E^I = v_1^I : \mathsf{Ty}(t_1) \dots v_k^I : \mathsf{Ty}(t_k)$ and $E^C = v_1^c : \mathsf{Ty}(\hat{t}_1) \dots v_n^c : \mathsf{Ty}(\hat{t}_n)$

$$\underline{\text{Formation}} \qquad \frac{\begin{array}{c} E \vdash t_j : \mathrm{s}_j \text{ for } j = 1 \dots k \\ E; E^I \vdash \hat{t}_i : \mathrm{s}'_i \text{ for } i = 1 \dots n \end{array}}{E \vdash \mathsf{IndSpec}(E^I := E^C) : \mathtt{blocktype}} \text{ if } \mathsf{POS}_E(E^I := E^C)$$

$$\underline{\text{Definition}} \qquad \frac{E \vdash \mathsf{IndSpec}(E^I := E^C) : \mathtt{blocktype}}{E \vdash \mathsf{Ind}(E^I := E^C) : \mathsf{IndSpec}(E^I := E^C)}$$

$$\underline{\text{Inductive Type}} \qquad \frac{E \vdash B : \mathsf{IndSpec}(E^I := E^C)}{E \vdash B.j : t_j} \text{ for } j = 1 \dots k$$

$$\underline{\text{Constructor}} \qquad \frac{E \vdash B : \mathsf{IndSpec}(E^I := E^C)}{E \vdash B.i : \hat{t}_{i-k}\{v_j^I \mapsto B.j\}_{j=1\dots k}} \text{ for } i = k+1 \dots n$$

The main role of an anonymous basic block $\mathsf{Ind}(E^I := E^C)$ is to provide an implementation for the inductive specification in a structure. It can of course also appear in terms as an anonymous inductive definition, using the the rules Inductive Type and Constructor.

The typing rules for rewrite blocks are similar to those given in Section 1.4 for definitions by rewriting. Let $E^\Gamma = v_1 : \mathsf{Ty}(t_1) \dots v_n : \mathsf{Ty}(t_n)$ and $R = \{E_i \mid e_i^L \longrightarrow e_i^R : \hat{t}_i\}_{i=1\dots m}$, where $E_i = v_1^i : \mathsf{Ty}(t_1^i); \dots ; v_{n_i}^i : \mathsf{Ty}(t_{n_i}^i)$

Formation

$$\frac{\begin{array}{c} E \vdash t_k : \mathrm{s}_k \quad \text{for } k = 1 \dots n \\ \left.\begin{array}{c} E; v_1^i : \mathsf{Ty}(t_1^i); \dots ; v_{j-1}^i : \mathsf{Ty}(t_{j-1}^i) \vdash t_j^i : \mathrm{s}_j^i \quad \text{for } j = 1 \dots n_i \\ E; E^\Gamma; E_i \vdash \hat{t}_i : \hat{s}_i \qquad E; E^\Gamma; E_i \vdash e_i^L : \hat{t}_i \qquad E; E^\Gamma; E_i \vdash e_i^R : \hat{t}_i \end{array}\right\} \text{for } i = 1 \dots m \end{array}}{E \vdash \mathsf{RewSpec}(E^\Gamma, R) : \mathtt{blocktype}}$$

$$\text{if } \ \mathsf{ACC}_E(E^\Gamma, R)$$

$$\underline{\text{Definition}} \qquad \frac{E \vdash \mathsf{RewSpec}(E^\Gamma, R) : \mathtt{blocktype}}{E \vdash \mathsf{Rew}(E^\Gamma, R) : \mathsf{RewSpec}(E^\Gamma, R)} \text{ if } \mathsf{COMP}_E(E^\Gamma, R)$$

The side condition $\mathsf{COMP}_E(E^\Gamma, R)$, is a *completeness condition.* It is supposed to assure that all functions from E^Γ are totally defined by R. The meaning of this condition is explained in the next subsection (paragraph "Rewriting in structures") and discussed in Chapter 11.

$$\underline{\text{Function}} \qquad \frac{E \vdash B : \mathsf{RewSpec}(E^\Gamma, R)}{E \vdash B.j : t_j}$$

$$\underline{\text{Computation}} \qquad \frac{E \vdash B : \mathsf{RewSpec}(E^\Gamma, R) \qquad \delta : E_i \to E}{E \vdash e_i^L\theta\delta \approx e_i^R\theta\delta} \quad \theta = \{v_j \mapsto B.j\}_{j=1\ldots n}$$

The notation $\delta : E_i \to E$ means that $E \vdash v\delta : t\delta$ holds for every $v : \mathsf{Ty}(t) \in E_i$.

$$\underline{\text{Satisfaction}} \qquad \frac{\begin{array}{c} E \vdash \mathsf{RewSpec}(E^\Gamma, R) : \texttt{blocktype} \\ E \vdash B.j : t_j \text{ for } j = 1\ldots n \\ E; E_i \vdash e_i^L\theta \approx e_i^R\theta \text{ for } i = 1\ldots m \end{array}}{E \vdash B : \mathsf{RewSpec}(E^\Gamma, R)} \quad \theta = \{v_j \mapsto B.j\}_{j=1\ldots n}$$

The Satisfaction rule is mainly useful for subtyping signatures containing rewrite specifications.

The typing rules for abstract and manifest block specifications are the same as for term and module specifications, so we do not repeat them here. It is enough to make ϕ a block B, Ψ a block type BT, and Ξ the block sort `blocktype`.

1.6.2 Characteristics of the calculus

The way to enter a rewrite specification in an environment is by first using the Formation rule for $\mathsf{RewSpec}()$, making sure it is correct

$$E \vdash \mathsf{RewSpec}(E^\Gamma, R) : \texttt{blocktype}$$

Then, using the abstract specification Formation rule, a correct specification is constructed

$$E \vdash \mathsf{Ty}(\mathsf{RewSpec}(E^\Gamma, R)) : \texttt{spec}$$

Finally, the environment is extended by using the environment rule Insert, yielding

$$E, \overline{v} : \mathsf{Ty}(\mathsf{RewSpec}(E^\Gamma, R)) \vdash \texttt{ok}$$

Adding an inductive definition to the environment is similar.

We now show how to use a rewrite specification from the environment. Let us call E' an environment which is an arbitrary extension of the one above. We assume that we have $E' \vdash \texttt{ok}$. The first step consists in using the environment Lookup rule to derive

$$E' \vdash \overline{v} : \mathsf{Ty}(\mathsf{RewSpec}(E^\Gamma, R))$$

Second, we apply the abstract specification Elimination rule, yielding

$$E' \vdash \overline{v} : \mathsf{RewSpec}(E^\Gamma, R)$$

Now we can either use the Function rule to get the type of some $v^j \in \overline{v}$ or the Computation rule to derive a convertibility judgment. Suppose R is our rewriting system defining *plus*. In that case $\overline{v}$ is a sequence of length 1. Suppose, we want to use the second rewrite rule from R, i.e. `plus (S x) y` $\Rightarrow$ `S (plus x y)`. To do that, we have to provide a well-typed substitution δ assigning well typed terms to all variables from the local environment of the rule $E_2 = x : \mathsf{Ty}(nat);\ y : \mathsf{Ty}(nat)$. Let $\delta = \{x \mapsto O; y \mapsto O\}$. The substitution δ is well-typed, because $E' \vdash O : nat$. Applying the Computation rule yields:

$$E' \vdash v\ (S\ O)\ O \approx S\ (v\ O\ O)$$

Note that our formal system does not permit to orient the equality obtained by the Computation rule. In presence of the Satisfaction rule, naive orientation from left to right would inevitably lead to nontermination. Let us consider the following example: let $E^\Gamma = f : \mathsf{Ty}(nat \to nat)$ and let $R_1 = \{f\ O \longrightarrow f\ (S\ O);\ f\ (S\ x) \longrightarrow O\}$ and $R_2 = \{f\ (S\ x) \longrightarrow f\ O;\ f\ O \longrightarrow O\}$ and suppose both rewriting systems satisfy the acceptance condition ACC. In the environment E containing $v : \mathsf{Ty}(\mathsf{RewSpec}(\Gamma, R_1))$, we have $E \vdash v : \mathsf{RewSpec}(E^\Gamma, R_1)$ by environment Lookup followed by abstract specification Elimination. Hence we can derive

$$E \vdash v\ O \approx v\ (S\ O)$$

On the other hand, using Satisfaction, it is possible to derive $E \vdash v : \mathsf{RewSpec}(E^\Gamma, R_2)$, because both equalities $E; x : \mathsf{Ty}(nat) \vdash v\ (S\ x) \approx v\ O$ and $E \vdash v\ O \approx O$ are satisfied. Hence we can derive

$$E \vdash v\ (S\ O) \approx v\ O$$

If both displayed equalities were oriented from left to right, we would have a loop.

It seems that the correct way of defining the convertibility relation by oriented reduction rules is via a derivation system similar to the one for δ-reduction (Definition 1.5.1). However, since we do not study the anonymous system in this thesis, we leave the question unanswered.

Rewriting in structures. Inside a structure, a definition by rewriting is composed of a sequence of labels, a sequence of binders, a specification and an implementation. The already mentioned informal structure defining addition on natural numbers:

```
Struct
  Symbol plus : nat → nat → nat
  Rules
    plus O y ⇒ y
    plus (S x) y ⇒ S (plus x y)
    plus x O ⇒ x
    plus x (S y) ⇒ S (plus x y)
    plus x (plus y z) ⇒ plus (plus x y) z
End
```

is represented in the formal syntax by the structure:

$$\mathsf{Struct}\ \mathtt{plus}\triangleright plus : \mathsf{Ty}(\mathsf{RewSpec}(E^\Gamma, R)) := \mathsf{Rew}(E^\Gamma, R)\ \mathsf{End}$$

whose signature is

$$\mathsf{Sig}\ \mathtt{plus}\triangleright plus : \mathsf{Ty}(\mathsf{RewSpec}(E^\Gamma, R))\ \mathsf{End}$$

where E^Γ and R are as before. The well-typedness derivation of this structure uses the Definition rule for rewrite specification and the completeness condition $\mathsf{COMP}_E(E^\Gamma, R)$ is necessarily verified. This condition, which is left unspecified like ACC and POS, is supposed to make sure that all functions declared in E^Γ and defined by the rewrite rules R are total.

The completeness condition permits to maintain the correspondence between structures and closed environments. In other words, the judgment $E \vdash \mathsf{RewSpec}(E^\Gamma, R)$: $\mathtt{blocktype}$, means that the specification is correct while the judgment $E \vdash \mathsf{Rew}(E^\Gamma, R)$: $\mathsf{RewSpec}(E^\Gamma, R)$ means that the specification is satisfiable.

From the practical point of view, the condition $\mathsf{ACC}_E(E^\Gamma, R)$ should imply decidability of type-checking in $E' = E, v : \mathsf{Ty}(\mathsf{RewSpec}(E^\Gamma, R))$, while the condition $\mathsf{COMP}_E(E^\Gamma, R)$ should imply the consistency of E', both properties relative with respect to E.

The completeness of definitions in presence of dependent types have been studied in [27]. The topic of completeness of definitions and consistency of the calculus with rewriting is not yet very well studied but there are already some results on this subject [12].

Generativity of definitions. The specifications $\mathsf{Ty}()$ and $\mathsf{Eq}()$ may be used to control the generativity of definitions. A definition of v is called generative, if it is impossible to write an expression equal to v, without explicitly mentioning the name v. We will illustrate this by examples.

In the first example, we discuss different specifications given for a module containing an inductive definition of natural numbers. To shorten the presentation, we write as usual

$$\begin{array}{lcl} E^I & = & nat : \mathsf{Ty}(\star) \\ E^C & = & 0 : \mathsf{Ty}(nat);\ S : \mathsf{Ty}(nat \to nat) \end{array}$$

Let us consider a structure m containing a definition of natural numbers (it will be given later) and let us now consider the following definitions, each having the same implementation m:

$$\begin{array}{lcl}
\mathtt{N}_0 & : & \mathsf{Ty}(\ \mathsf{Sig}\ \ \mathtt{nat} : \mathsf{Ty}(\star);\ \mathtt{0} : \mathsf{Ty}(nat);\ \mathtt{S} : \mathsf{Ty}(nat \to nat)\ \ \mathsf{End}\ ) := m \\
\mathtt{N}_1 & : & \mathsf{Ty}(\ \mathsf{Sig}\ \ \mathtt{nat}, \mathtt{0}, \mathtt{S} : \mathsf{Ty}(\mathsf{IndSpec}(E^I := E^C))\ \ \mathsf{End}\ ) := m \\
\mathtt{N}'_1 & : & \mathsf{Ty}(\ \mathsf{Sig}\ \ \mathtt{nat}, \mathtt{0}, \mathtt{S} : \mathsf{Ty}(\mathsf{IndSpec}(E^I := E^C))\ \ \mathsf{End}\ ) := m \\
\mathtt{N}_2 & : & \mathsf{Ty}(\ \mathsf{Sig}\ \ \mathtt{nat}, \mathtt{0}, \mathtt{S} : \mathsf{Eq}(\mathsf{Ind}(E^I := E^C) : \mathsf{IndSpec}(E^I := E^C))\ \ \mathsf{End}\ ) := m \\
\mathtt{N}'_2 & : & \mathsf{Ty}(\ \mathsf{Sig}\ \ \mathtt{nat}, \mathtt{0}, \mathtt{S} : \mathsf{Eq}(\mathsf{Ind}(E^I := E^C) : \mathsf{IndSpec}(E^I := E^C))\ \ \mathsf{End}\ ) := m
\end{array}$$

In the first definition, $N_0.\mathtt{nat}$ is not declared as an inductive type. We only know that it is a type. For the next two, $N_1.\mathtt{nat}$ and $N'_1.\mathtt{nat}$ are two generative inductive definitions. Because their respective specifications do not tell what they are equal to, each of them is only equal to itself. Therefore, $N_1.\mathtt{nat} \not\approx N'_1.\mathtt{nat}$. For the last two, $N_2.\mathtt{nat}$ and $N'_2.\mathtt{nat}$ are two non-generative inductive types, both convertible with $\mathsf{Ind}(E^I := E^C).1$ and therefore with each other.

In order to satisfy all of the above specifications the structure m must be defined as follows:

$$\mathsf{Struct}\ \ \mathtt{nat}, \mathtt{0}, \mathtt{S}\ :\ \mathsf{Eq}(\mathsf{Ind}(E^I := E^C) : \mathsf{IndSpec}(E^I := E^C))\ := \ \mathsf{Ind}(E^I := E^C)\ \ \mathsf{End}$$

Analogous variations of specifications are also possible for definitions by rewriting. Let us again use the example of *plus* and the structure m defining it:

$$\mathsf{Struct}\ \ \mathtt{plus} : \mathsf{Eq}(\mathsf{Rew}(E^\Gamma, R) : \mathsf{RewSpec}(E^\Gamma, R)\ := \ \mathsf{Rew}(E^\Gamma, R)\ \ \mathsf{End}$$

as well as five possible specifications given to this implementation:

$$\begin{array}{l}
\mathtt{M}_0 : \mathsf{Ty}(\ \mathsf{Sig}\ \mathtt{plus} : \mathsf{Ty}(nat \to nat \to nat)\ \ \mathsf{End}\ )\ :=\ m \\
\mathtt{M}_1 : \mathsf{Ty}(\ \mathsf{Sig}\ \mathtt{plus} : \mathsf{Ty}(\mathsf{RewSpec}(E^\Gamma, R))\ \mathsf{End}\ )\ :=\ m \\
\mathtt{M}'_1 : \mathsf{Ty}(\ \mathsf{Sig}\ \mathtt{plus} : \mathsf{Ty}(\mathsf{RewSpec}(E^\Gamma, R))\ \mathsf{End}\ )\ :=\ m \\
\mathtt{M}_2 : \mathsf{Ty}(\ \mathsf{Sig}\ \mathtt{plus} : \mathsf{Eq}(\mathsf{Rew}(E^\Gamma, R) : \mathsf{RewSpec}(E^\Gamma, R))\ \mathsf{End}\ )\ :=\ m \\
\mathtt{M}'_2 : \mathsf{Ty}(\ \mathsf{Sig}\ \mathtt{plus} : \mathsf{Eq}(\mathsf{Rew}(E^\Gamma, R) : \mathsf{RewSpec}(E^\Gamma, R))\ \mathsf{End}\ )\ :=\ m
\end{array}$$

Suppose there are three variables a, b and c of type nat. We will consider the term $M.\mathtt{plus}\ a\ (M.\mathtt{plus}\ b\ c)$, with M substituted subsequently by M_0, M_1, …, M'_2. As with inductive types, the first specification is too restrictive and consequently the term $M_0.\mathtt{plus}\ a\ (M_0.\mathtt{plus}\ b\ c)$ is only equal to itself. When substituting any of the other four specifications to M, the term $M.\mathtt{plus}\ a\ (M.\mathtt{plus}\ b\ c)$ is equal to $M.\mathtt{plus}\ (M.\mathtt{plus}\ a\ b)\ c$. Now, consider $M_1.\mathtt{plus}\ a\ (M'_1.\mathtt{plus}\ b\ c)$. Since both definitions are generative, this term is only equal to itself. On the other hand, because M_2 and M'_2 have manifest specifications, $M_2.\mathtt{plus}\ a\ (M'_2.\mathtt{plus}\ b\ c)$ is convertible with $\mathsf{Rew}(E^\Gamma, R).1\ a\ (\mathsf{Rew}(E^\Gamma, R).1\ b\ c)$ and therefore with $\mathsf{Rew}(E^\Gamma, R).1\ (\mathsf{Rew}(E^\Gamma, R).1\ a\ b)\ c$.

Rewriting and functors Rewrite specifications used in the parameter interface of a functor may be used to impose convertibility constraints on the functor arguments. Before presenting the actual example, let us suppose that $P : nat \to \star$ is a predicate, and let us define the following two abbreviations:

$$\begin{array}{rcl}
BT & = & \mathsf{RewSpec}(f : \mathsf{Ty}(nat \to nat);\ g : \mathsf{Ty}(nat \to nat), \\
 & & \qquad n : \mathsf{Ty}(nat) \mid f\ (g\ n) \longrightarrow g\ (f\ n)\ : nat\ \ ) \\
B & = & \mathsf{Rew}(\mathtt{h} : \mathsf{Ty}(nat \to nat),\ \ m : nat \mid h\ m \longrightarrow S(S\,m) : nat)
\end{array}$$

Let us now consider the following functor:

$$\begin{array}{l}
\mathsf{Functor}(X\ :\ \mathsf{Sig}\ \mathtt{f},\mathtt{g} :\ \mathsf{Ty}(BT)\ \mathsf{End}) \\
\quad \mathsf{Struct} \\
\qquad \mathtt{lemma} : \mathsf{Ty}(\ (n : nat)P(X.\mathtt{f}(X.\mathtt{f}(X.\mathtt{g}\ n))) \to P(X.\mathtt{g}(X.\mathtt{f}(X.\mathtt{f}\ n)))\ ) \\
\qquad\qquad := [n : nat][p : P(X.\mathtt{f}(X.\mathtt{f}(X.\mathtt{g}\ n)))]\ p \\
\quad \mathsf{End}
\end{array}$$

In the functor body, type-checking the implementation of its single `lemma` component relies on the rewrite rule placed in the parameter interface. Thanks to the notion of subtyping, the functor presented above can be applied to the structure

$$\begin{array}{l}
\mathsf{Struct} \\
\quad \mathtt{g} :\ \mathsf{Eq}(\ [n : nat](S\,n) : nat \to nat\ ) := [n : nat](S\,n) \\
\quad \mathtt{f}\ :\ \mathsf{Eq}(\ B.1 : nat \to nat\ ) := B.1 \\
\mathsf{End}
\end{array}$$

In fact, the Satisfaction typing rule allows any sequence of function symbols to satisfy the specification $\mathsf{RewSpec}(E^\Gamma, R)$ as long as the corresponding conversion can prove equalities of left-hand and right-hand sides of all rules in R. In our example, the structure defines the functions f and g in such a way that both $f(g\,n)$ and $g(f\,n)$ are convertible with $S(S(S\,n))$, and therefore the equation expressed by the rule is satisfied.

1.6.3 Closure properties for rewriting

This section illustrates how reductions on anonymous module expressions may interfere with definitions by rewriting. Consider the following functor

$$\begin{array}{l} \mathsf{Functor}[X : \mathsf{Sig}\ \mathsf{a} : \mathsf{Ty}(nat)\ \mathsf{End}] \\ \quad \mathsf{Struct} \\ \qquad \mathtt{f} : \mathsf{Ty}(\ \mathsf{RewSpec}(f : \mathsf{Ty}(nat \to nat),\ \ x\!:\!\mathsf{Ty}(nat) \mid f\ x \longrightarrow X.\mathsf{a}\ \ : nat)\ ) := \ldots \\ \quad \mathsf{End} \end{array}$$

Its application to the structure $\mathsf{Struct}\ \mathsf{a} : \mathsf{Eq}(O : nat) := O\ \mathsf{End}$ β_m-reduces to

$$\begin{array}{l} \mathsf{Struct} \\ \quad \mathtt{f} : \mathsf{Ty}(\ \mathsf{RewSpec}(f : \mathsf{Ty}(nat \to nat), \\ \qquad\qquad\qquad x\!:\!\mathsf{Ty}(nat) \mid\ f\ x \longrightarrow \mathsf{Struct}\ \mathsf{a} : \mathsf{Eq}(O : nat) := O\ \mathsf{End}.\mathsf{a}\ \ : nat\ )\ ) := \ldots \\ \mathsf{End} \end{array}$$

which in turn, since a has a manifest specification, reduces to

$$\begin{array}{l} \mathsf{Struct} \\ \quad \mathtt{f} : \mathsf{Ty}(\ \mathsf{RewSpec}(f : \mathsf{Ty}(nat \to nat),\ \ x\!:\!\mathsf{Ty}(nat) \mid f\ x \longrightarrow O\ \ : nat)\ ) := \ldots \\ \mathsf{End} \end{array}$$

Note that if a were itself defined by rewriting, the reduction would lead to a nested definition by rewriting, quite an unusual feature.

In general, the subject reduction property implies that the conditions ACC() and COMP() for accepting the rewrite rules must be closed under substitution and reduction. These closure properties unfortunately do not hold for the currently known methods for proving termination or confluence [3, 12, 99]. One must therefore either restrict the left-hand and right-hand sides of the rules in such a way that substitutions and reductions are impossible or put limitations on the calculus in order to relax the closure requirements for the conditions on rewrite rules. In the remaining part of this thesis, we adopt the second possibility.

Note also that this is not the anonymous definitions by rewriting per se which impose these strong closure properties on the acceptance conditions, but the fact that definitions by rewriting occur inside anonymous modules. The simplest way to limit the impact of the module calculus on the definitions by rewriting is to get rid of anonymous modules and their reductions.

1.7 Named modules

The calculus studied in this thesis is obtained by imposing several syntactic restrictions on the calculus of anonymous modules presented in the previous section. The most important restriction is the necessity to *name* all complex module expressions before using them in terms. In this way, the language of terms will not contain the general module selection expression $m.\mathtt{v}$, but only so called *access paths* consisting of possibly nested selections of a field from a module variable, i.e. $v.\mathtt{v}_1.\cdots.\mathtt{v}_n$. This restriction is actually present in many module systems used in functional programming languages (see for example [59]).

Another restriction is the elimination of anonymous inductive types and anonymous definitions by rewriting. From a syntactic point of view, we do not allow expressions like $B.i$ where B is a block, and we allow only sequences of access paths in manifest block specifications, as in $\mathsf{Eq}(\overline{p} : BT)$ where $\overline{p}$ denotes a sequence of access paths. Consequently, anonymous block expressions $\mathsf{Ind}()$ and $\mathsf{Rew}()$ are only allowed as implementations of their respective specifications inside structures. Inductive definitions and definitions by rewriting become therefore generative, and behave in the same way as the ones presented in Sections 1.3 and 1.4.

As a consequence of these syntactic restrictions, there are no more module reductions. A β_m-redex is not even a correct module expression, as the application of modules is restricted to application of access paths.

1.8 Contributions

In this thesis we show that adding named modules, as briefly described in the previous section, to a pure type system with generative inductive definitions and definitions by rewriting does not compromise the system's consistency and decidability of type-checking.

Assuming that consistency of a calculus with generative definitions means that no closed formal development proving `False` exists, the conservativity of the modular extension is proved by a procedure transforming any closed development (with modules) into a closed development without modules.

The procedure has two phases: first, all module components of the formal development are *evaluated* to a weak-head normal form using the call-by-value evaluation strategy; and second, the resulting evaluated development is *flattened*, i.e. components of modules are promoted to the toplevel, and modules themselves removed.

Since the module language itself forms a typed lambda calculus, termination of the first phase is not immediate. It is shown by translating our module expressions to a weak version of the module calculus of Courant and relating our evaluation steps with reductions between translated expressions. Since the calculus of Courant is terminating, so is our evaluation.

Proving the correctness of the second phase requires showing that module typing rules (subtyping in particular) are not essential when typing terms. We have therefore defined a simplified calculus, called *principal*, which contains only typing rules for terms and rules to extract term information from the environment. We show that term judgments derivable in both calculi are the same, a result which can also be interpreted as soundness of module subtyping. In the principal system, showing that module flattening preserves typing is an easy induction on the derivation.

The relative decidability proof is based on the observation that type preservation for flattening works in both directions: a flattened term is typable in a flattened environment if and only if the original term is typable in the original environment. Since the flattening operation itself has polynomial complexity, this gives us a polynomial reduction of the problem of typing terms in an arbitrary environment to the typing problem for the principal system in an environment without modules. And since derivations in the latter system correspond to derivations in a PTS with generative definitions, we have the desired relative decidability result.

We also prove that our module language has the principal typing property, which implies decidability of the module typing itself as well, again, under the assumption that typing in the base PTS is decidable.

As we explained in Section 1.4, in order to assure that a (consistent and decidable) pure type system extended with rewriting is consistent and decidable, it is necessary to restrict the class of rewriting systems acceptable in a correct environment. Therefore we assume the existence of an *acceptance condition*, which acts as a guard of the environment's correctness: every rewriting system is verified against this condition before being added to the environment and in return the acceptance condition guarantees that type-checking in the correct environment is decidable.

In our thesis, we do not fix any particular acceptance condition. We only list closure properties which have to be verified by such a condition in order to make our proofs work. These properties may be summarized as modularity and substitutivity of names by names and they are compatible with acceptance conditions studied in the literature [3, 11, 99]. In Chapter 11 we discuss our closure properties in detail and propose a condition satisfying them.

Let us remark that our closure properties are much weaker than those which would be necessary for the system with anonymous modules of Courant. In particular we do not need closure by arbitrary substitutions, closure by name substitution (i.e. substituting names to names) being enough. The latter property is much simpler to fulfill in practice, as name substitutions do not change the syntactic form of terms on which syntactic termination criteria for rewriting systems rely [3, 12, 99].

Similarly, we do not impose any particular positivity criterion for inductive definitions. They have to verify the same closure properties as acceptance conditions for rewrite rules, but this is much less problematic.

The main practical result of this thesis is the implementation of the module system in the Coq proof assistant.[14] Apart from implementing a module type-checker in the kernel of Coq, a considerable implementation effort has been done towards the modularization

[14] Definitions by rewriting have not yet been implemented however. The experimental "Rewriting" branch of Coq is developed by Frédéric Blanqui, but is not yet provided in the distributed versions of Coq.

of extra-logical features of Coq such as databases for automated tactics, customizable parsing and pretty-printing, etc. Part of this thesis describing implementation of modules in Coq has been accepted for presentation at the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs) in Rome, Italy, 9-12 September 2003 [19].

Modules are included in the version 7.4 of Coq, officially released in February 2003. Since then, several large developments using modules have been realized, proving their usefulness in proof development.

1.9 Roadmap

The remaining part of this thesis is organized as follows. Chapter 2 contains the definition of our module system: its syntax, typing rules, closure properties imposed on block formation rules (Section 2.7.1) and auxiliary definitions such as positions or bound variables.

Chapter 3 formulates the main result of this thesis, i.e. conservativity (Theorem 1), and informally describes the proof method which takes four consecutive chapters.

Chapter 4 studies basic metatheoretical properties of our calculus such as substitutivity by access paths and correctness of module types, block types and specifications assigned to expressions.

Chapter 5 introduces a restricted version of typing, called principal, which contains only term typing rules and elimination rules for modules and blocks. It is shown that module paths have principal types and that principal derivations are complete in a sense that all term judgments derivable in the full system are also derivable in the principal system.

Chapter 6 contains the details of first part of the proof of Theorem 1. It shows how to evaluate module applications without breaking syntactic restrictions on paths. The call-by-value evaluation is defined and shown to be type preserving and terminating.

Chapter 7 proves the second part of the conservativity theorem, called *flattening*. It strongly relies on properties of principal derivations.

Chapter 8 shows the reduction of the typing problem for the calculus with modules and manifest blocks to typing in the calculus without modules and without manifest blocks (Theorem 2), showing that modular extension of a decidable theory is decidable.

Chapter 9 describes the implementation of the module system in Coq. Various implementation problems are presented together with adopted solutions.

Chapter 10 presents related work on module systems in ML family of programming languages and structuring facilities in other proof assistants.

Chapter 11 discusses closure properties for acceptance conditions for definitions by rewriting, their informal motivation, practical meaning and proposes acceptance conditions satisfying them.

The last chapter mentions some developments realized with modules and presents future research topics and directions.

Chapter 2

Named modules

In this chapter we give the complete mathematical definition of the calculus of named modules implemented in the Coq version 7.4 and studied in this thesis. We start by the syntax, then we give the typing rules and definitions of auxiliary concepts like positions and bound variables.

Our system is a syntactic restriction of the anonymous calculus presented in Section 1.6 in the introduction. In a few words: all modules are named and all inductive and rewrite definitions are generative.

2.1 Syntax

Since our system is an extension of a PTS, we start by giving a set of sorts $\mathcal{S}$, a set of PTS axioms $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ and a set of product rules $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$. We assume that $\star$ is an element of $\mathcal{S}$. This assumption will be used in Section 6.3.

- Access path:

$p ::= v$	variable
$\mid p.\mathtt{v}$	field selection

- Term[1]:

$e, t ::= p$	term path
$\mid \mathrm{s}$	$\mathrm{s} \in \mathcal{S}$, a PTS sort
$\mid (e_1\ e_2)$	application
$\mid [v{:}t]\ e$	abstraction
$\mid (v{:}t_1)\ t_2$	product

- Term specification:

$TS ::= \mathsf{Ty}(t)$	abstract

[1]The two letters e and t correspond to the same syntactic class. The letter t is used in places where a term intuitively plays a role of a type.

$\mid \mathsf{Eq}(e : t)$ manifest

- Module expression:

$m ::= \mathsf{Struct}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n := P_n\ \mathsf{End}$ structure
$\mid \mathsf{Functor}[v{:}M]\ m$ functor
$\mid p$ module path
$\mid (p_1\ p_2)$ path application

- Module type:

$M ::= \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$ module signature
$\mid \mathsf{Funsig}(v{:}M_1)\ M_2$ functor type

- Module specification:

$MS ::= \mathsf{Ty}(M)$ abstract
$\mid \mathsf{Eq}(p : M)$ manifest

- Basic block:

$BB ::= \mathsf{Ind}(E^I := E^C)$ inductive definition
$\mid \mathsf{Rew}(E^\Gamma, R)$ definition by rewriting

- Block:

$B ::= BB$ basic block
$\mid \overline{p}$ sequence of paths

- Block type:

$BT ::= \mathsf{IndSpec}(E^I := E^C)$ inductive specification
$\mid \mathsf{RewSpec}(E^\Gamma, R)$ rewrite specification

- Block specification:

$BS ::= \mathsf{Ty}(BT)$ abstract
$\mid \mathsf{Eq}(\overline{p} : BT)$ manifest

- Specification:

$S ::= TS \mid MS \mid BS$

- Implementation:

$P ::= e \mid m \mid B$

- Environment:

$E ::= \epsilon \mid \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n$

- Rewriting System:

$R ::= \{E_i \mid e_i^L \longrightarrow e_i^R : t_i\}_{i=1\ldots n}$

A rewriting system is a sequence of rules. We use the syntax with braces for the sake of brevity. An element of either syntactic class presented above besides environments and rewriting systems (E and R) is called an *expression*. An element which belongs to either of the main syntactic classes presented above or auxiliary syntactic classes presented below is called an *extended expression*.

The following syntactic classes are used to abbreviate the formulation of some definitions and lemmas:

- Structure Component:
  $StrC ::= \overline{\mathsf{v}} \triangleright \overline{v} : S := P$

- Structure Component Sequence:
  $StrCs ::= StrC_1 \ldots StrC_n$

- Signature Component:
  $SigC ::= \overline{\mathsf{v}} \triangleright \overline{v} : S$

- Signature Component Sequence:
  $SigCs ::= SigC_1 \ldots SigC_n$

- Environment Component:
  $EnvC ::= \overline{v} : S$

- Rewrite Rule:
  $Rule ::= E \mid e^L \longrightarrow e^R : t$

Metavariables in this thesis are always meaningful, i.e. "*For all m such that...*" should always be read as "*For all module expressions m such that...*" etc. Metavariables different from those listed above are treated less strictly. In particular ψ, ϕ are used to mean expressions, i.e. terms e, modules m or blocks B. The symbols Ψ, Φ denote types: terms t, module types M or block types BT. The symbol Ξ denotes a sort: a PTS sort s, the module sort `modtype` or the block sort `blocktype`. These symbols are often used in judgments and rules concerning the specifications of the three kinds: term, module and block.

The auxiliary syntactic function $length(\cdot)$ is equal to the number of terms in blocks, block types and block specifications. It is naturally extended to specifications and environments.

- Blocks
  $length(\mathsf{Rew}(E^\Gamma, R)) = length(E^\Gamma)$
  $length(\mathsf{Ind}(E^I := E^C)) = length(E^I) + length(E^C)$
  $length(p^1 \ldots p^n) = n$

- Block types
  $length(\mathsf{RewSpec}(E^\Gamma, R)) = length(E^\Gamma)$
  $length(\mathsf{IndSpec}(E^I := E^C)) = length(E^I) + length(E^C)$

- Block specifications
  $length(\mathsf{Ty}(BT)) = length(BT)$
  $length(\mathsf{Eq}(B : BT)) = length(BT)$

- Term specifications
  $length(TS) = 1$

- Module specifications
  $length(MS) = 1$

- Environments
  $length(\overline{v}_1 : S_1 \ldots \overline{v}_n : S_n) = length(\overline{v}_1) + \cdots + length(\overline{v}_n)$

Given a structure m, we will use the abbreviation $SIG(m)$ to denote a *primary signature* of the structure m, defined as follows:

$$SIG(\mathsf{Struct}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n := P_n\ \mathsf{End}) \ = \ \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$$

2.2 Judgments

There are many forms of judgments in our calculus. We present here the complete list of them, together with the intuitive meaning:

1. $E \vdash \mathtt{ok}$ — the environment E is well-formed.

2. $E \vdash \Phi_1 \approx \Phi_2$ — in the environment E, Φ_1 and Φ_2 are convertible.

   The possibilities for Φ_1 and Φ_2 are the following:

   - $E \vdash e_1 \approx e_2$ — terms,
   - $E \vdash p_1 \approx p_2$ — access paths,
   - $E \vdash BT_1 \approx BT_2$ — block types,
   - $E \vdash \overline{p}_1 \approx \overline{p}_2$ — path sequences,
   - $E \vdash M_1 \approx M_2$ — module types,
   - $E \vdash S_1 \approx S_2$ — specifications

3. $E \vdash M_1 <: M_2$ — in the environment E the module type M_1 is a subtype of M_2

4. $E \vdash \phi : \Psi$ — in the environment E, ϕ can be ascribed with Ψ.

   The possibilities for ϕ and Ψ are the following:

- $E \vdash e : t$ term e has type t
- $E \vdash e : TS$ term e verifies term specification TS
- $E \vdash m : M$ module expression m has module type M
- $E \vdash M : \texttt{modtype}$ module type M is well-formed
- $E \vdash m : MS$ module expression m verifies module specification MS,
- $E \vdash B : BT$ block B has block type BT
- $E \vdash BT : \texttt{blocktype}$ block type BT is correct
- $E \vdash B : BS$ block B verifies block specification BS
- $E \vdash S : \texttt{spec}$ specification S is well-formed

The judgment $E \vdash \overline{p}_1 \approx \overline{p}_2$ should be read as an abbreviation for a sequence of judgments $E \vdash p_1^1 \approx p_2^1 \ldots E \vdash p_1^n \approx p_2^n$.

The judgments $E \vdash M_1 \approx M_2$ and $E \vdash S_1 \approx S_2$ are not used in any conversion rule. They are only used in proofs.

It is possible to classify our judgments differently, into three main categories of terms, modules and blocks and two auxiliary ones: the environment correctness judgment $E \vdash \texttt{ok}$ and specification correctness judgment $E \vdash S : \texttt{spec}$. Each of the main three categories have ascription judgments $E \vdash \phi : \Psi$ and convertibility judgments $E \vdash \phi_1 \approx \phi_2$. Ascription judgments are: an object has a type, a type has a sort (is correct), an object verifies a specification. Convertibility judgments are for objects and for their types. In case of terms the first two ascription judgments collapse, as well as the convertibility judgments. Modules have one additional judgment concerning subtyping, and their object convertibility is reduced to access paths. Of course the typing judgments like $E \vdash M : \texttt{modtype}$ (or $E \vdash BT : \texttt{blocktype}$ or $E \vdash S : \texttt{spec}$) could have been presented as specialized judgments of the form $E \vdash \texttt{well-formed-module-type}(M)$ but we chose the typing-like presentation. In case of `modtype` and `blocktype` it makes it easier to write rules for specifications and in the case of `spec` it is an arbitrary decision.

2.3 Environment

Variables in environments are assumed to be pairwise disjoint, so in the second rule $\overline{v}$ are fresh variables and $length(\overline{v}) = length(S)$.

(Env/Empty) $$\frac{}{\epsilon \vdash \texttt{ok}}$$

(Env/Insert) $$\frac{E \vdash \texttt{ok} \qquad E \vdash S : \texttt{spec}}{E; \overline{v} : S \vdash \texttt{ok}}$$

(Env/Lookup) $$\frac{E_1; \overline{v} : S; E_2 \vdash \texttt{ok}}{E_1; \overline{v} : S; E_2 \vdash \overline{v} : S}$$

2.4 Terms

The rules in this section correspond to the typing rules for pure type systems.

$$\textbf{(Term/Prod)} \qquad \frac{E \vdash t_1 : \mathrm{s}_1 \quad E; v : \mathsf{Ty}(t_1) \vdash t_2 : \mathrm{s}_2}{E \vdash (v{:}t_1)\ t_2 : \mathrm{s}_3} \text{ if } (\mathrm{s}_1, \mathrm{s}_2, \mathrm{s}_3) \in \mathcal{R}$$

$$\textbf{(Term/Abs)} \qquad \frac{E; v : \mathsf{Ty}(t_1) \vdash e : t_2 \quad E \vdash (v{:}t_1)\ t_2 : \mathrm{s}}{E \vdash [v{:}t_1]e : (v{:}t_1)\ t_2}$$

$$\textbf{(Term/Axiom)} \qquad \frac{E \vdash \mathtt{ok}}{E \vdash \mathrm{s}_1 : \mathrm{s}_2} \text{ if } (\mathrm{s}_1, \mathrm{s}_2) \in \mathcal{A}$$

$$\textbf{(Term/App)} \qquad \frac{E \vdash e : (v{:}t_1)\ t_2 \quad E \vdash e' : t_1}{E \vdash e\ e' : t_2\{v \mapsto e'\}}$$

$$\textbf{(Term/Beta)} \qquad [v{:}t]e\ e' \longrightarrow_\beta e\{v \mapsto e'\}$$

$$\textbf{(Term/Conv)} \qquad \frac{E \vdash e : t \qquad E \vdash t' : \mathrm{s} \qquad E \vdash t \approx t'}{E \vdash e : t'}$$

2.5 Modules

The module rules are almost identical to those defined in [32] and presented in Section 1.5. Only the rules for signature access and module application are restricted to paths. Also, since module expressions are not part of terms any more and hence are not concerned by the conversion rule, module β_m reduction is no longer needed.

Structures and signatures

$$\textbf{(Sig/Form)} \qquad \frac{E; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n \vdash \mathtt{ok}}{E \vdash \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End} : \mathtt{modtype}}$$

if $\overline{\mathtt{v}}_1 \ldots \overline{\mathtt{v}}_n$ are pairwise disjoint labels
and $\mathit{length}(\overline{\mathtt{v}}_i) = \mathit{length}(\overline{v}_i)$ for all i

(Sig/Struct)

$$\frac{\begin{array}{c} E \vdash \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End} : \mathtt{modtype} \\ E; \overline{v}_1 : S_1 \ldots \overline{v}_{k-1} : S_{k-1} \vdash P_k : S_k \quad \text{for } k = 1 \ldots n \end{array}}{E \vdash \mathsf{Struct}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n := P_n\ \mathsf{End} : \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}}$$

The access rule is restricted to module paths.

$$\textbf{(Sig/Access)} \qquad \frac{E \vdash p : \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}}{E \vdash \overline{p.\mathtt{v}_k} : S_k\{v_i \mapsto p.\mathtt{v}_i\}_{i=1 \ldots k-1}}$$

(Sig/Sub)

$$\frac{\begin{array}{c} E \vdash \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End} : \mathtt{modtype} \\ E \vdash \mathsf{Sig}\ \overline{\mathtt{v}}'_1 \triangleright \overline{v}'_1 : S'_1 \ldots \overline{\mathtt{v}}'_{n'} \triangleright \overline{v}'_{n'} : S'_{n'}\ \mathsf{End} : \mathtt{modtype} \\ \{\overline{\mathtt{v}}'_1 \ldots \overline{\mathtt{v}}'_{n'}\} \subseteq \{\overline{\mathtt{v}}_1 \ldots \overline{\mathtt{v}}_n\} \\ \text{by } \alpha\text{-conversion, we assume } \mathtt{v}'^{j'}_{i'} = \mathtt{v}^j_i \text{ implies } v'^{j'}_{i'} = v^j_i \\ E; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n \vdash \overline{v}'_k : S'_k \quad \text{for } k = 1 \ldots n' \end{array}}{E \vdash \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}\ <:\ \mathsf{Sig}\ \overline{\mathtt{v}}'_1 \triangleright \overline{v}'_1 : S'_1 \ldots \overline{\mathtt{v}}'_{n'} \triangleright \overline{v}'_{n'} : S'_{n'}\ \mathsf{End}}$$

Functors and functor types Functors are exactly like in ML-like module systems. They correspond to functions and their types to products. They can be higher order and their subtyping is contravariant in the argument, but covariant in the output type.

(Funsig/Form) $$\frac{E \vdash M_1 : \mathtt{modtype} \qquad E; v : \mathsf{Ty}(M_1) \vdash M_2 : \mathtt{modtype}}{E \vdash \mathsf{Funsig}(v{:}M_1)\ M_2 : \mathtt{modtype}}$$

(Funsig/Functor) $$\frac{E; v : \mathsf{Ty}(M_1) \vdash m : M_2 \qquad E \vdash \mathsf{Funsig}(v{:}M_1) M_2 : \mathtt{modtype}}{E \vdash \mathsf{Functor}[v{:}M_1]\ m : \mathsf{Funsig}(v{:}M_1) M_2}$$

(Funsig/Sub) $$\frac{E \vdash M'_1 <: M_1 \quad E; v : \mathsf{Ty}(M'_1) \vdash M_2 <: M'_2}{E \vdash \mathsf{Funsig}(v{:}M_1)\ M_2 <: \mathsf{Funsig}(v{:}M'_1)\ M'_2}$$

Like the access rule, the application rule is restricted to paths.

(Mod/App) $$\frac{E \vdash p : \mathsf{Funsig}(v : M_1)\ M_2 \qquad E \vdash p' : M_1}{E \vdash p\ p' : M_2\{v \mapsto p'\}}$$

The computation rule β_m is no longer needed.

Other rules

(Subsumption) $$\frac{E \vdash m : M_1 \quad E \vdash M_1 <: M_2}{E \vdash m : M_2}$$

(Strengthening) $$\frac{E \vdash p : M}{E \vdash p : M/p}$$

where the strengthening operation $\cdot/p$ is defined for module types, specifications, block types and terms as follows:

$$\begin{array}{rcl} \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}/p & = & \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1/\overline{p.\mathtt{v}_1}; \ldots ; \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n/\overline{p.\mathtt{v}_n}\ \mathsf{End} \\ (\mathsf{Funsig}(v{:}M_1)\ M_2)/p & = & \mathsf{Funsig}(v{:}M_1)\ M_2 \\ \mathsf{Eq}(\overline{\phi} : \Psi)/\overline{p} & = & \mathsf{Eq}(\overline{\phi} : \Psi/\overline{p}) \\ \mathsf{Ty}(\Psi)/\overline{p} & = & \mathsf{Eq}(\overline{p} : \Psi/\overline{p}) \\ BT/\overline{p} & = & BT \\ t/p & = & t \end{array}$$

2.6 Specifications

The following rules define abstract and manifest specifications. An abstract specification $\mathsf{Ty}(\Psi)$ asserts that a given object has type Ψ (or more generally: properties defined by Ψ). A manifest specification $\mathsf{Eq}(\phi : \Psi)$ asserts that a given object has type Ψ and is equal to ϕ. The formation rule guarantees of course that ϕ itself of type Ψ. Since the rules for abstract/manifest specifications are the same for terms, modules and blocks we give them only once. In the following rules triples ϕ, Ψ, Ξ are e, t, s or $p, M, \mathtt{modtype}$ or $\overline{p}, BT, \mathtt{blocktype}$.

Abstract specifications

$$\textbf{(Ty/Form)} \qquad \frac{E \vdash \Psi : \Xi}{E \vdash \mathsf{Ty}(\Psi) : \mathtt{spec}}$$

$$\textbf{(Ty/Sat)} \qquad \frac{E \vdash P : \Psi \qquad E \vdash \mathsf{Ty}(\Psi) : \mathtt{spec}}{E \vdash P : \mathsf{Ty}(\Psi)}$$

$$\textbf{(Ty/Type)} \qquad \frac{E \vdash P : \mathsf{Ty}(\Psi)}{E \vdash P : \Psi}$$

Manifest specifications

$$\textbf{(Eq/Form)} \qquad \frac{E \vdash \phi : \Psi \qquad E \vdash \Psi : \Xi}{E \vdash \mathsf{Eq}(\phi : \Psi) : \mathtt{spec}}$$

$$\textbf{(Eq/Sat)} \qquad \frac{E \vdash P \approx \phi \qquad E \vdash P : \Psi \qquad E \vdash \mathsf{Eq}(\phi : \Psi) : \mathtt{spec}}{E \vdash P : \mathsf{Eq}(\phi : \Psi)}$$

$$\textbf{(Eq/Type)} \qquad \frac{E \vdash P : \mathsf{Eq}(\phi : \Psi)}{E \vdash P : \Psi}$$

$$\textbf{(Eq/Comp)} \qquad \frac{E \vdash P : \mathsf{Eq}(\phi : \Psi)}{E \vdash P \approx \phi}$$

2.7 Blocks

Block rules are almost identical to those defined is Section 1.6. The only difference is that rules for block type satisfaction, elimination and computation are restricted to path sequences.

Inductive blocks Let $E^I = v_1^I : \mathsf{Ty}(t_1) \ldots v_k^I : \mathsf{Ty}(t_k)$, $\quad E^C = v_1^C : \mathsf{Ty}(\hat{t}_1) \ldots v_n^C : \mathsf{Ty}(\hat{t}_n)$

$$\textbf{(IndSpec/Form)} \qquad \frac{\begin{array}{c} E \vdash t_j : \mathrm{s}_j \text{ for } j = 1 \ldots k \\ E; E^I \vdash \hat{t}_i : \mathrm{s}'_i \text{ for } i = 1 \ldots n \end{array}}{E \vdash \mathsf{IndSpec}(E^I := E^C) : \mathtt{blocktype}} \text{ if } \mathsf{POS}_E(E^I := E^C)$$

$$\textbf{(IndSpec/Ind)} \qquad \frac{E \vdash \mathsf{IndSpec}(E^I := E^C) : \mathtt{blocktype}}{E \vdash \mathsf{Ind}(E^I := E^C) : \mathsf{IndSpec}(E^I := E^C)}$$

$$\textbf{(IndSpec/IndType)} \qquad \frac{E \vdash \overline{p} : \mathsf{IndSpec}(E^I := E^C)}{E \vdash p^j : t_j} \text{ for } j = 1 \ldots k$$

$$\textbf{(IndSpec/Constr)} \qquad \frac{E \vdash \overline{p} : \mathsf{IndSpec}(E^I := E^C)}{E \vdash p^{k+i} : \hat{t}_i\{v_j^I \mapsto p^j\}_{j=1\ldots k}} \text{ for } i = 1 \ldots n$$

Rewrite blocks In the forthcoming rules $E^\Gamma = v_1 : \mathsf{Ty}(t_1) \ldots v_n : \mathsf{Ty}(t_n)$ and $R = \{E_i \mid e_i^L \longrightarrow e_i^R : \hat{t}_i\}_{i=1\ldots m}$, where $E_i = v_1^i : \mathsf{Ty}(t_1^i); \ldots ; v_{n_i}^i : \mathsf{Ty}(t_{n_i}^i)$

(RewSpec/Form)

$$\frac{\begin{array}{c} E \vdash t_k : \mathrm{s}_k \quad \text{for } k = 1 \ldots n \\ \left.\begin{array}{c} E; v_1^i : \mathsf{Ty}(t_1^i); \ldots ; v_{j-1}^i : \mathsf{Ty}(t_{j-1}^i) \vdash t_j^i : \mathrm{s}_j^i \quad \text{for } j = 1 \ldots n_i \\ E; E^\Gamma; E_i \vdash \hat{t}_i : \hat{s}_i \qquad E; E^\Gamma; E_i \vdash e_i^L : \hat{t}_i \qquad E; E^\Gamma; E_i \vdash e_i^R : \hat{t}_i \end{array}\right\} \text{for } i = 1 \ldots m \end{array}}{E \vdash \mathsf{RewSpec}(E^\Gamma, R) : \mathtt{blocktype}}$$

$$\text{if } \mathsf{ACC}_E(E^\Gamma, R)$$

$$\textbf{(RewSpec/Rew)} \qquad \frac{E \vdash \mathsf{RewSpec}(E^\Gamma, R) : \mathtt{blocktype}}{E \vdash \mathsf{Rew}(E^\Gamma, R) : \mathsf{RewSpec}(E^\Gamma, R)} \text{ if } \mathsf{COMP}_E(E^\Gamma, R)$$

$$\textbf{(RewSpec/Fun)} \qquad \frac{E \vdash \overline{p} : \mathsf{RewSpec}(E^\Gamma, R)}{E \vdash p^j : t_j}$$

(RewSpec/Comp)

$$\frac{E \vdash \overline{p} : \mathsf{RewSpec}(E^\Gamma, R) \qquad \delta : E_i \to E}{E \vdash e_i^L \theta\delta \approx e_i^R \theta\delta} \text{ where } \quad \theta = \{v_j \mapsto p^j\}_{j=1\ldots n}$$

The notation $\delta : E_i \to E$ means that $E \vdash v\delta : t\delta$ holds for every $v : \mathsf{Ty}(t) \in E_i$.

$$\textbf{(RewSpec/Sat)} \qquad \frac{\begin{array}{c} E \vdash \mathsf{RewSpec}(E^\Gamma, R) : \mathtt{blocktype} \\ E \vdash p^j : t_j \text{ for } j = 1 \ldots n \\ E; E_i \vdash e_i^L \theta \approx e_i^R \theta \text{ for } i = 1 \ldots m \end{array}}{E \vdash \overline{p} : \mathsf{RewSpec}(E^\Gamma, R)} \text{ where } \quad \theta = \{v_j \mapsto p^j\}_{j=1\ldots n}$$

Block conversion

$$\textbf{(Block/Conv)} \qquad \frac{E \vdash B : BT \qquad E \vdash BT' : \mathtt{blocktype} \qquad E \vdash BT \approx BT'}{E \vdash B : BT'}$$

2.7.1 Closure properties

Each one of the side-conditions C present in the above rules (POS, ACC and COMP) must satisfy the following closure properties:

$$\mathsf{C}_E(\Omega) \qquad \text{implies} \qquad \mathsf{C}_{E'}(\Omega') \qquad \text{if:}$$

(C1) $E = E_1; \overline{v} : S; E_2$ $\quad$ $E' = E_1; E_2\{\overline{v} \mapsto \overline{p}\}$ $\quad$ with $\quad E_1 \vdash \overline{p} : S$
Ω $\quad$ $\Omega' = \Omega\{\overline{v} \mapsto \overline{p}\}$,

(C2) $E = E_1; E_3$ $\quad$ $E' = E_1; E_2; E_3$
Ω $\quad$ $\Omega' = \Omega$

(C3) $E = E_1; v : \mathsf{Ty}(\mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}); E_2$
Ω
$E' = E_1; \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n; E_2\{\overline{v.\mathtt{v}_i} \mapsto \overline{v}_i\}$
$\Omega' = \Omega\{\overline{v.\mathtt{v}_i} \mapsto \overline{v}_i\}$

(C4) $E = E_1; v : \mathsf{Ty}(\mathsf{Funsig}(\hat{v} : M)M'); E_2$
Ω
$E' = E_1; E_2$
$\Omega' = \Omega$

(C5) $E = E_1; v : \mathsf{Eq}(p : M); E_2$ $\quad$ $E' = E_1; v : \mathsf{Ty}(M/p); E_2$
Ω $\quad$ $\Omega' = \Omega$

In all of the above implications it is assumed that both environments E and E' are correct. Moreover in the last three properties, E_2 is an environment without modules.

In short, these properties require the side-conditions to be modular and closed under arbitrary path substitutions. They are discussed in details in Chapter 11.

2.8 Convertibility

Convertibility is a congruence based on the equality judgments derivable from the rules **(Eq/Comp)** and **(RewSpec/Comp)** and from the rule **(Conv/Beta)** below. All other rules are needed to make it a congruence. The convertibility relation is used in typing derivations through the rules **(Block/Conv)** and **(Term/Conv)**. There are no such rules for module types and specification, as they can be deduced from others.

General

(Conv/Trans) $\dfrac{E \vdash \phi_1 \approx \phi_2 \quad E \vdash \phi_2 \approx \phi_3}{E \vdash \phi_1 \approx \phi_3}$ $\qquad$ **(Conv/Sym)** $\dfrac{E \vdash \phi_1 \approx \phi_2}{E \vdash \phi_2 \approx \phi_1}$

where ϕ_1, ϕ_2, ϕ_3 are terms or paths.

Paths

$$\textbf{(Conv/Refl)} \qquad \frac{E \vdash v : M}{E \vdash v \approx v}$$

$$\textbf{(Conv/Path)} \qquad \frac{E \vdash p.\mathtt{v} : \Psi \quad E \vdash p \approx p' \quad E \vdash p'.\mathtt{v} : \Psi'}{E \vdash p.\mathtt{v} \approx p'.\mathtt{v}}$$

where Ψ and Ψ' are module types or terms.

Terms

$$\textbf{(Conv/Beta)} \qquad \frac{E \vdash e_1 : t \quad e_1 =_\beta e_2 \quad E \vdash e_2 : t'}{E \vdash e_1 \approx e_2}$$

$$\textbf{(Conv/App)} \qquad \frac{E \vdash e_1\, e_2 : t \quad E \vdash e_1 \approx e_1' \quad E \vdash e_2 \approx e_2' \quad E \vdash e_1'\, e_2' : t'}{E \vdash e_1\, e_2 \approx e_1'\, e_2'}$$

(Conv/Prod)

$$\frac{E \vdash (v{:}t_1)\ t_2 : t \quad E \vdash t_1 \approx t_1' \quad E; v : \mathsf{Ty}(t_1) \vdash t_2 \approx t_2' \quad E \vdash (v{:}t_1')\ t_2' : t'}{E \vdash (v{:}t_1)\ t_2 \approx (v{:}t_1')\ t_2'}$$

(Conv/Abs)

$$\frac{E \vdash [v{:}t]e : \hat{t} \quad E \vdash t \approx t' \quad E; v : \mathsf{Ty}(t) \vdash e \approx e' \quad E \vdash [v{:}t']e' : \hat{t}'}{E \vdash [v{:}t]e \approx [v{:}t']e'}$$

Block types

$$\textbf{(Conv/IndSpec)} \qquad \frac{\begin{array}{c} E \vdash \mathsf{IndSpec}(E^I := E^C) : \mathtt{blocktype} \\ E \vdash t_j \approx t_j' \quad v_j^I = v^{I'}{}_j \quad \text{for } j = 1 \ldots k \\ E; E^I \vdash \hat{t}_i \approx \hat{t}_i{}' \quad v_i^C = v^{C'}{}_i \quad \text{for } i = 1 \ldots n \\ E \vdash \mathsf{IndSpec}(E^{I'} := E^{C'}) : \mathtt{blocktype} \end{array}}{E \vdash \mathsf{IndSpec}(E^I := E^C) \approx \mathsf{IndSpec}(E^{I'} := E^{C'})}$$

where $E^I = v_1^I : \mathsf{Ty}(t_1) \ldots v_k^I : \mathsf{Ty}(t_k) \quad E^{I'} = v_1^I : \mathsf{Ty}(t_1') \ldots v_k^I : \mathsf{Ty}(t_k')$
$E^C = v_1^C : \mathsf{Ty}(\hat{t}_1) \ldots v_n^C : \mathsf{Ty}(\hat{t}_n) \quad E^{C'} = v_1^C : \mathsf{Ty}(\hat{t'}_1) \ldots v_n^C : \mathsf{Ty}(\hat{t'}_n)$

(Conv/RewSpec)

$$\frac{\begin{array}{c} E \vdash \mathsf{RewSpec}(E^\Gamma, R) : \mathtt{blocktype} \\ E \vdash t_k \approx t'_k \quad \text{for } k = 1 \ldots n \\ \left.\begin{array}{c} E; v^i_1 : \mathsf{Ty}(t^i_1); \ldots ; v^i_{j-1} : \mathsf{Ty}(t^i_{j-1}) \vdash t^i_j \approx t'^i_j \quad \text{for } j = 1 \ldots n_i \\ E; E^\Gamma; E_i \vdash e^L_i \approx e^{L'}_i \qquad E; E^\Gamma; E_i \vdash e^R_i \approx e^{R'}_i \qquad E; E^\Gamma; E_i \vdash \hat{t}_i \approx \hat{t}_i{}' \end{array}\right\} \text{for } i = 1 \ldots m \\ E \vdash \mathsf{RewSpec}(E^{\Gamma\prime}, R') : \mathtt{blocktype} \end{array}}{E \vdash \mathsf{RewSpec}(E^\Gamma, R) \approx \mathsf{RewSpec}(E^{\Gamma\prime}, R')}$$

where $E^\Gamma = v_1 : \mathsf{Ty}(t_1) \ldots v_n : \mathsf{Ty}(t_n) \quad E^{\Gamma\prime} = v_1 : \mathsf{Ty}(t'_1) \ldots v_n : \mathsf{Ty}(t'_n)$
$R = \{E_i \mid e^L_i \longrightarrow e^R_i : \hat{t}_i\}_{i=1\ldots m} \quad R' = \{E'_i \mid e^{L'}_i \longrightarrow e^{R'}_i : \hat{t}_i{}'\}_{i=1\ldots m}$
$E_i = v^i_1 : \mathsf{Ty}(t^i_1) \ldots v^i_{n_i} : \mathsf{Ty}(t^i_{n_i}) \quad E'_i = v^i_1 : \mathsf{Ty}(t^i_1{}') \ldots v^i_{n_i} : \mathsf{Ty}(t^i_{n_i}{}')$

Specifications The judgments $E \vdash S_1 \approx S_2$ and $E \vdash M_1 \approx M_2$ are not used in conversion rules. They are only used in proofs.

(Conv/Ty) $$\frac{E \vdash \mathsf{Ty}(\Psi) : \mathtt{spec} \quad E \vdash \Psi \approx \Psi' \quad E \vdash \mathsf{Ty}(\Psi') : \mathtt{spec}}{E \vdash \mathsf{Ty}(\Psi) \approx \mathsf{Ty}(\Psi')}$$

(Conv/Eq)

$$\frac{E \vdash \mathsf{Eq}(\phi : \Psi) : \mathtt{spec} \quad E \vdash \phi \approx \phi' \quad E \vdash \Psi \approx \Psi' \quad E \vdash \mathsf{Eq}(\phi' : \Psi') : \mathtt{spec}}{E \vdash \mathsf{Eq}(\phi : \Psi) \approx \mathsf{Eq}(\phi' : \Psi')}$$

Module types

(Conv/Sig) $$\frac{\begin{array}{c} E \vdash \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End} : \mathtt{modtype} \\ E; \overline{v}_1 : S_1 \ldots \overline{v}_{i-1} : S_{i-1} \vdash S_i \approx S'_i \ \text{ for } i = 1 \ldots n \\ E \vdash \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S'_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S'_n\ \mathsf{End} : \mathtt{modtype} \end{array}}{E \vdash \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End} \approx \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S'_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S'_n\ \mathsf{End}}$$

(Conv/Funsig) $$\frac{\begin{array}{c} E \vdash \mathsf{Funsig}(v{:}M_1)\ M_2 : \mathtt{modtype} \\ E \vdash M_1 \approx M'_1 \quad E; v : \mathsf{Ty}(M_1) \vdash M_2 \approx M'_2 \\ E \vdash \mathsf{Funsig}(v{:}M'_1)\ M'_2 : \mathtt{modtype} \end{array}}{E \vdash \mathsf{Funsig}(v{:}M_1)\ M_2 \approx \mathsf{Funsig}(v{:}M'_1)\ M'_2}$$

2.9 Positions, subexpressions and bound variables

In this section we define positions and subexpressions of an expression.

Definition 2.9.1. *The set of positions of an extended expression ψ, denoted by $Pos(\psi)$, is defined as follows (the left column is to indicate the syntactic class of a given ψ):*

$$
\begin{array}{lrcl}
p & Pos(v) & = & \{\epsilon\} \\
 & Pos(p.\mathtt{v}) & = & \{\epsilon\} \cup 1 \cdot Pos(p) \\
e, t & Pos(\mathrm{s}) & = & \{\epsilon\} \\
 & Pos(e_1\ e_2) & = & \{\epsilon\} \cup 1 \cdot Pos(e_1) \cup 2 \cdot Pos(e_2) \\
 & Pos([v{:}t]\ e) & = & \{\epsilon\} \cup 1 \cdot Pos(t) \cup 2 \cdot Pos(e) \\
 & Pos((v{:}t_1)\ t_2) & = & \{\epsilon\} \cup 1 \cdot Pos(t_1) \cup 2 \cdot Pos(t_2) \\
S & Pos(\mathsf{Ty}(\Psi)) & = & \{\epsilon\} \cup 1 \cdot Pos(\Psi) \\
 & Pos(\mathsf{Eq}(\phi : \Psi)) & = & \{\epsilon\} \cup 1 \cdot Pos(\phi) \cup 2 \cdot Pos(\Psi) \\
m & \multicolumn{3}{l}{Pos(\mathsf{Struct}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n := P_n\ \mathsf{End}) =} \\
 & \multicolumn{3}{r}{\{\epsilon\} \cup 1 \cdot Pos(\overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1) \cup \cdots \cup n \cdot Pos(\overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n := P_n)} \\
 & Pos(\mathsf{Functor}[v{:}M]\ m) & = & \{\epsilon\} \cup 1 \cdot Pos(M) \cup 2 \cdot Pos(m) \\
 & Pos(p_1\ p_2) & = & \{\epsilon\} \cup 1 \cdot Pos(p_1) \cup 2 \cdot Pos(p_2) \\
StrC & Pos(\overline{\mathtt{v}} \triangleright \overline{v} : S := P) & = & \{\epsilon\} \cup 1 \cdot Pos(S) \cup 2 \cdot Pos(P) \\
M & \multicolumn{3}{l}{Pos(\mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}) =} \\
 & \multicolumn{3}{r}{\{\epsilon\} \cup 1 \cdot Pos(\overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1) \cup \cdots \cup n \cdot Pos(\overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n)} \\
 & Pos(\mathsf{Funsig}(v{:}M_1)\ M_2) & = & \{\epsilon\} \cup 1 \cdot Pos(M_1) \cup 2 \cdot Pos(M_2) \\
SigC & Pos(\overline{\mathtt{v}} \triangleright \overline{v} : S) & = & \{\epsilon\} \cup 1 \cdot Pos(S) \\
B & Pos(\mathsf{Ind}(E^I := E^C)) & = & \{\epsilon\} \cup 1 \cdot Pos(E^I) \cup 2 \cdot Pos(E^C) \\
 & Pos(\mathsf{Rew}(E^\Gamma, R)) & = & \{\epsilon\} \cup 1 \cdot Pos(E^\Gamma) \cup 2 \cdot Pos(R) \\
 & Pos(p^1 \ldots p^n) & = & \{\epsilon\} \cup 1 \cdot Pos(p_1) \cup \cdots \cup n \cdot Pos(p_n) \\
BT & Pos(\mathsf{IndSpec}(E^I := E^C)) & = & \{\epsilon\} \cup 1 \cdot Pos(E^I) \cup 2 \cdot Pos(E^C) \\
 & Pos(\mathsf{RewSpec}(E^\Gamma, R)) & = & \{\epsilon\} \cup 1 \cdot Pos(E^\Gamma) \cup 2 \cdot Pos(R) \\
E & Pos(\overline{v}_1 : S_1 \ldots \overline{v}_n : S_n) & = & \{\epsilon\} \cup 1 \cdot Pos(\overline{v}_1 : S_1) \cup \cdots \cup n \cdot Pos(\overline{v}_n : S_n) \\
EnvC & Pos(\overline{v} : S) & = & \{\epsilon\} \cup 1 \cdot Pos(S) \\
R & \multicolumn{3}{l}{Pos(\{E_j \mid e_j^L \longrightarrow e_j^R : t_j\}_{j=1\ldots n}) =} \\
 & \multicolumn{3}{r}{\{\epsilon\} \cup 1 \cdot Pos(E_1 \mid e_1^L \longrightarrow e_1^R : t_1) \cup \cdots \cup n \cdot Pos(E_n \mid e_n^L \longrightarrow e_n^R : t_n)} \\
Rule & \multicolumn{3}{l}{Pos(E \mid e^L \longrightarrow e^R : t) =} \\
 & \multicolumn{3}{r}{\{\epsilon\} \cup 1 \cdot Pos(E) \cup 2 \cdot Pos(e^L) \cup 3 \cdot Pos(e^R) \cup 4 \cdot Pos(t)}
\end{array}
$$

where $i \cdot A = \{i \cdot w \mid w \in A\}$

If one treats an extended expression ψ like a labelled tree, the skeleton of which is the set of positions $Pos(\psi)$, then the subtree rooted at position q is denoted by $\psi|_q$. Those subtrees are called *subexpressions*.

Definition 2.9.2. *Given an extended expression ψ and a position $q \in Pos(\psi)$, the environment of* bound variables *of ψ at position q, denoted by $BV(\psi, q)$, is defined as follows:*

$$
\begin{array}{lrcl}
p & BV(p,\ q) & = & \epsilon \\
e, t & BV(\mathrm{s},\ q) & = & \epsilon \\
 & BV(e_1\ e_2,\ 1 \cdot q) & = & BV(e_1, q) \\
 & BV(e_1\ e_2,\ 2 \cdot q) & = & BV(e_2, q) \\
 & BV([v{:}t]\ e,\ 1 \cdot q) & = & BV(t, q) \\
 & BV([v{:}t]\ e,\ 2 \cdot q) & = & v : \mathsf{Ty}(t); BV(e, q) \\
 & BV((v{:}t_1)\ t_2,\ 1 \cdot q) & = & BV(t_1, q) \\
 & BV((v{:}t_1)\ t_2,\ 2 \cdot q) & = & v : \mathsf{Ty}(t_1); BV(t_2, q) \\
S & BV(\mathsf{Ty}(\Psi),\ 1 \cdot q) & = & BV(\Psi, q) \\
 & BV(\mathsf{Eq}(\phi : \Psi),\ 1 \cdot q) & = & BV(\phi, q) \\
 & BV(\mathsf{Eq}(\phi : \Psi),\ 2 \cdot q) & = & BV(\Psi, q) \\
m & \multicolumn{3}{l}{BV(\mathsf{Struct}\ \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n := P_n\ \mathsf{End},\ i \cdot q) =} \\
 & \multicolumn{3}{r}{\overline{v}_1 : S_1 \ldots \overline{v}_{i-1} : S_{i-1}; BV(\overline{\mathsf{v}}_i \triangleright \overline{v}_i : S_i := P_i,\ q)} \\
 & BV(\mathsf{Functor}[v{:}M]\ m,\ 1 \cdot q) & = & BV(M, q) \\
 & BV(\mathsf{Functor}[v{:}M]\ m,\ 2 \cdot q) & = & v : \mathsf{Ty}(M); BV(m, q) \\
 & BV(p_1\ p_2,\ q) & = & \epsilon \\
StrC & BV(\overline{\mathsf{v}} \triangleright \overline{v} : S := P,\ 1 \cdot q) & = & BV(S, q) \\
 & BV(\overline{\mathsf{v}} \triangleright \overline{v} : S := P,\ 2 \cdot q) & = & BV(P, q) \\
M & BV(\mathsf{Sig}\ \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End};\ i \cdot q) & = & \overline{v}_1 : S_1 \ldots \overline{v}_{i-1} : S_{i-1}; BV(\overline{\mathsf{v}}_i \triangleright \overline{v}_i : S_i,\ q) \\
 & BV(\mathsf{Funsig}(v{:}M_1)\ M_2,\ 1 \cdot q) & = & BV(M_1; q) \\
 & BV(\mathsf{Funsig}(v{:}M_1)\ M_2,\ 2 \cdot q) & = & v : \mathsf{Ty}(M); BV(M_2, q) \\
SigC & BV(\overline{\mathsf{v}} \triangleright \overline{v} : S,\ 1 \cdot q) & = & BV(S, q) \\
B & BV(\mathsf{Ind}(E^I := E^C),\ 1 \cdot q) & = & BV(E^I, q) \\
 & BV(\mathsf{Ind}(E^I := E^C),\ 2 \cdot q) & = & E^I; BV(E^C, q) \\
 & BV(\mathsf{Rew}(E^\Gamma, R),\ 1 \cdot q) & = & BV(E^\Gamma, q) \\
 & BV(\mathsf{Rew}(E^\Gamma, R),\ 2 \cdot q) & = & E^\Gamma; BV(R, q) \\
 & BV(\overline{p},\ q) & = & \epsilon \\
BT & BV(\mathsf{IndSpec}(E^I := E^C),\ 1 \cdot q) & = & BV(E^I, q) \\
 & BV(\mathsf{IndSpec}(E^I := E^C),\ 2 \cdot q) & = & E^I; BV(E^C, q) \\
 & BV(\mathsf{RewSpec}(E^\Gamma, R),\ 1 \cdot q) & = & BV(E^\Gamma, q) \\
 & BV(\mathsf{RewSpec}(E^\Gamma, R),\ 2 \cdot q) & = & E^\Gamma; BV(R, q) \\
E & BV(\overline{v}_1 : S_1 \ldots \overline{v}_n : S_n,\ i \cdot q) & = & \overline{v}_1 : S_1 \ldots \overline{v}_{i-1} : S_{i-1}; BV(\overline{v}_i : S_i,\ q) \\
EnvC & BV(\overline{v} : S, 1 \cdot q) & = & BV(S, q)
\end{array}
$$

$$
\begin{array}{ll}
R & BV(\{E_j \mid e_j^L \longrightarrow e_j^R : t_j\}_{j=1\ldots n},\ i \cdot q) = BV(E_i \mid e_i^L \longrightarrow e_i^R : t_i,\ q) \\
Rule & BV(E \mid e^L \longrightarrow e^R : t,\ 1 \cdot q) = BV(E, q) \\
 & BV(E \mid e^L \longrightarrow e^R : t,\ 2 \cdot q) = E; BV(e^L, q) \\
 & BV(E \mid e^L \longrightarrow e^R : t,\ 3 \cdot q) = E; BV(e^R, q) \\
 & BV(E \mid e^L \longrightarrow e^R : t,\ 4 \cdot q) = E; BV(t, q)
\end{array}
$$

Chapter 3

Conservativity roadmap

We are going to show that our calculus of named modules is conservative with respect to the pure type system with inductive types and rewriting. Imposing conditions on rewriting that guarantee logical consistency of the whole typing system is a difficult task which is out of the scope of this thesis (see discussion in Chapter 11). We will therefore implicitly assume that developments in the pure type system with inductive types and rewriting are consistent.

The natural idea of reusing the conservativity proof of Courant does not work, because his proof uses the subject reduction property with respect to modules reductions, which do not exist in our calculus. It would even be impossible to define such reductions since they violate our syntactic restrictions about access paths. Nevertheless, we are going to simulate module reductions following a call-by-value strategy and show that modules can be eliminated from arbitrary developments.

3.1 Conservativity

We argue that a closed Coq development (without axioms) corresponds to a structure typable in the empty environment, where components of the structure are only manifest or abstract term definitions, for constants and lemmas respectively, and inductive definitions. Since we want to show conservativity with respect to the calculus with rewriting, we permit also definitions by rewriting. To make a difference from general structure components, we call the former ones classic.

Definition 3.1.1. *A structure component is* classic *if it is of one of the following forms:*

1. $\mathsf{v} \triangleright v : \mathsf{Ty}(t) := e$,
2. $\mathsf{v} \triangleright v : \mathsf{Eq}(e : t) := e$,
3. $\overline{\mathsf{v}} \triangleright \overline{v} : \mathsf{Ty}(\mathsf{IndSpec}(E^I := E^C)) := \mathsf{Ind}(E^I := E^C)$ *or*
4. $\overline{\mathsf{v}} \triangleright \overline{v} : \mathsf{Ty}(\mathsf{RewSpec}(E^\Gamma, R)) := \mathsf{Rew}(E^\Gamma, R)$.

Classic signature components are defined similarly.

Without losing generality, we may assume that a development in Coq can be divided into three parts: modelisation, proof development and final theorem. The modelisation corresponds to writing definitions of constants (components of the form 2) and inductive definitions (of the form 3). If rewriting were already available in Coq, definitions by rewriting (of the form 4) would also participate to the modelisation phase. The proof development corresponds to writing components of any kind — these are auxiliary definitions and lemmas — and the final theorem is a component of the form 1. For example, let us imagine proving commutativity of addition on natural numbers. The modelisation part would contain the definition of natural numbers and addition, the proving phase would contain auxiliary lemmas (like the fact that zero is a left and right neutral element for addition) and the final theorem would state and prove the commutativity of addition.

Definition 3.1.2. *A* formal development *is a structure well-typed in the empty environment, consisting of the following three parts:*

- modelisation — *a sequence of classic structure components*

  $\overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n := P_n$

- proof development — *a sequence of (any) structure components*

  $\overline{\mathsf{v}}_{n+1} \triangleright \overline{v}_{n+1} : S_{n+1} := P_{n+1} \ldots \overline{\mathsf{v}}_m \triangleright \overline{v}_m : S_m := P_m$

- final theorem — *a structure component*

  $\mathsf{v} \triangleright v : \mathsf{Ty}(t) := e$

Moreover, the term t, called goal *is well-typed in the modelisation environment, i.e.*

$$\overline{v}_1 : S_1 \ldots \overline{v}_n : S_n \vdash t : \star$$

A formal development is classic *if its proof development part contains only classic components.*

The conservativity result to be proved in this thesis shows that no new theorems can be proved using modules and manifest blocks.

Theorem 1. *Every formal development can be transformed into a classic formal development with the same modelisation and the same goal.*

In other words, the fact we have to prove is the following:

Let $\overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n := P_n$ be a sequence of classic components and t a term such that $\overline{v}_1 : S_1 \ldots \overline{v}_n : S_n \vdash t : \star$. If there exists a structure

$$\begin{array}{l}
\mathsf{Struct} \\
\quad \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n := P_n \\
\quad \overline{\mathsf{v}}_{n+1} \triangleright \overline{v}_{n+1} : S_{n+1} := P_{n+1} \ldots \overline{\mathsf{v}}_m \triangleright \overline{v}_m : S_m := P_m \\
\quad \mathsf{v} \triangleright v : \mathsf{Ty}(t) := e \\
\mathsf{End}
\end{array}$$

well-typed is the empty environment, then there exists a classic structure of the form

$$\begin{array}{l}\mathsf{Struct}\\ \quad \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n := P_n\\ \quad \overline{\mathtt{v}}'_{n+1} \triangleright \overline{v}'_{n+1} : S'_{n+1} := P'_{n+1} \ldots \overline{\mathtt{v}}'_k \triangleright \overline{v}'_k : S'_k := P'_k\\ \quad \mathtt{v} \triangleright v : \mathsf{Ty}(t) := e'\\ \mathsf{End}\end{array}$$

well-typed in the empty environment.

The proof of this theorem is done in two phases. First, we *evaluate* all module components in the initial structure from left to right, and then we eliminate modules from right to left, by simply removing functors and *flattening* structures. Once all functors are removed and all sub-structures are flattened, we are left with a classic structure.

In the rest of this chapter we give a more detailed presentation of the two phases.

3.2 Evaluation

In this phase, we are going to evaluate module components to a weak head normal form.

Definition 3.2.1. *A module expression m is in* weak head normal form *if it is a path, a structure or a functor*[1]*.*

Definition 3.2.2. *A structure component is* evaluated *if it is a term or block component or if it is a module component $\mathtt{v} \triangleright v : \mathsf{Ty}(M) := m$ such that*

- *m is in weak head normal form,*
- *if m is a structure then it is evaluated and M is the primary signature of m*

A structure is evaluated if all its components are.

Lemma 3.2.1. *Let $\overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n := P_n$ be a sequence of classic components and t a term such that $\overline{v}_1 : S_1 \ldots \overline{v}_n : S_n \vdash t : \star$. If there exists a structure*

$$\begin{array}{l}\mathsf{Struct}\\ \quad \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n := P_n\\ \quad \overline{\mathtt{v}}_{n+1} \triangleright \overline{v}_{n+1} : S_{n+1} := P_{n+1} \ldots \overline{\mathtt{v}}_m \triangleright \overline{v}_m : S_m := P_m\\ \quad \mathtt{v} \triangleright v : \mathsf{Ty}(t) := e\\ \mathsf{End}\end{array}$$

[1] but not an application

well-typed is the empty environment, then there exists an evaluated structure of the form

$$\begin{array}{l}\mathsf{Struct}\\ \quad \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n := P_n\\ \quad \overline{\mathtt{v}}_{n+1} \triangleright \overline{v}_{n+1} : S'_{n+1} := P'_{n+1} \ldots \overline{\mathtt{v}}_m \triangleright \overline{v}_m : S'_m := P'_m\\ \quad \mathtt{v} \triangleright v : \mathsf{Ty}(t) := e\\ \mathsf{End}\end{array}$$

well-typed in the empty environment.

The proof of this lemma is given in Chapter 6, the final result of which is Lemma 6.3.21. It works by evaluating the structure components from left to right. The evaluation operates on module components only, in the following way:

- structures are evaluated recursively and are finally given the primary signature, as follows:

  $\mathtt{v} \triangleright v : \mathsf{Ty}(M) := \mathsf{Struct}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n := P_n\ \mathsf{End} \quad \xrightarrow{eval}{}^{*}$

  $\mathtt{v} \triangleright v : \mathsf{Ty}(M) := \mathsf{Struct}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : \hat{S}_1 := \hat{P}_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : \hat{S}_n := \hat{P}_n\ \mathsf{End} \quad \xrightarrow{eval}$

  $\mathtt{v} \triangleright v : \mathsf{Ty}(\mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : \hat{S}_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : \hat{S}_n\ \mathsf{End})$
  $\quad := \ \mathsf{Struct}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : \hat{S}_1 := \hat{P}_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : \hat{S}_n := \hat{P}_n\ \mathsf{End}$

- functor applications are evaluated in the following way:

  $\mathtt{v} \triangleright v : \mathsf{Ty}(M) := p_1\ p_2 \quad \xrightarrow{eval}$

  $\mathtt{v} \triangleright v : \mathsf{Ty}(M) := m\{\hat{v} \mapsto p_2\}$ if the value of p_1 is $\mathsf{Functor}[\hat{v} : M']m$

  and

  $\mathtt{v} \triangleright v : \mathsf{Ty}(M) := p_1\ p_2 \quad \xrightarrow{eval}$

  $\mathtt{v} \triangleright v : \mathsf{Ty}(M) := p'\ p_2$ otherwise, assuming the value of p_1 is p'

The value of a path in the given context is defined recursively as follows:

- if the path is a variable v then the value of v is the m part of the component $\mathtt{v} \triangleright v : \mathsf{Ty}(M) := m$ occurring in the context,
- if the path is of the form $p.\mathtt{v}_i$ then:
  - if the value of p is a structure

    $$\mathsf{Struct}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \mathtt{v}_i \triangleright v_i : \mathsf{Ty}(M) := m \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n := P_n\ \mathsf{End}$$

    then the value of $p.\mathtt{v}_i$ is $m\{\overline{v}_j \mapsto \overline{p.\mathtt{v}_j}\}$.
  - otherwise, assuming the value of p is p', the value of $p.\mathtt{v}_i$ is $p'.\mathtt{v}_i$.

The fact that the value of a path is well defined follows from the type correctness of the considered structure and from the fact that the preceding components have already been evaluated.

The evaluation process is terminating, which can be proved using the translation to the weak version of the calculus of Judicaël Courant, $\mathcal{WMC}$. This calculus is used by Courant to show decidability of module typing under the simplifying assumption that typing of terms is trivial: for all terms e and t we have $E \vdash_W e : t$.

The translation of our calculus to $\mathcal{WMC}$ is straightforward. We translate all block and term components by a sequence of trivial components $\mathtt{v} \triangleright v : \mathsf{Eq}(\star : \star) := \star$. The translation of module components is done by making all specifications manifest.

It turns out that evaluation of a path application corresponds to actual module reductions in $\mathcal{WMC}$, performed on the translated expressions. Since $\mathcal{WMC}$ is terminating, this guarantees termination of the evaluation of a single module component.

Unfortunately module type modification performed after recursively evaluating a substructure does not correspond to a $\mathcal{WMC}$ reduction of the translated expression, so the termination of the whole evaluation process cannot be deduced directly. However, by carefully analyzing the normal forms of the translation to $\mathcal{WMC}$ before and after the module type modification, we may notice that the number of module components in both normal forms is the same. Consequently the total number of module components evaluated during the whole process is bounded and therefore the evaluation cannot go on forever.

We also prove that each evaluation step preserves the type of the whole structure. In the end, starting from a structure m of type M, we obtain an evaluated structure m' of type M, and classic components of m are not changed in the evaluation process.

3.3 Flattening

The second phase of transforming a given structure into a classic one consists in removing the module components (the proof is given in Chapter 7 in Lemma 7.2.5):

Lemma 3.3.1. *Let $\overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n := P_n$ be a sequence of classic components and t a term such that $\overline{v}_1 : S_1 \ldots \overline{v}_n : S_n \vdash t : \star$. If there exists an evaluated structure*

$$
\begin{array}{l}
\mathsf{Struct} \\
\quad \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n := P_n \\
\quad \overline{\mathtt{v}}_{n+1} \triangleright \overline{v}_{n+1} : S'_{n+1} := P'_{n+1} \ldots \overline{\mathtt{v}}_m \triangleright \overline{v}_m : S'_m := P'_m \\
\quad v : \mathsf{Ty}(t) := e \\
\mathsf{End}
\end{array}
$$

well-typed is the empty environment, then there exists a classic structure of the form

$$\begin{array}{l} \mathsf{Struct} \\ \quad \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n := P_n \\ \quad \overline{\mathtt{v}}'_{n+1} \triangleright \overline{v}'_{n+1} : S'_{n+1} := P'_{n+1} \ldots \overline{\mathtt{v}}'_k \triangleright \overline{v}'_k : S'_k := P'_k \\ \quad \mathtt{v} \triangleright v : \mathsf{Ty}(t) := e' \\ \mathsf{End} \end{array}$$

well-typed is the empty environment.

We proceed here from right to left, eliminating the last non-classic component in the following way:

- if the last non-classic component is a module component:

  $\mathtt{v} \triangleright v : \mathsf{Ty}(\mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End})$
  $\quad := \ \mathsf{Struct}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n := P_n\ \mathsf{End}$

  then we replace it by its sub-components, i.e:

  $\overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n := P_n$

  and perform the inverse substitution $\{\overline{v.\mathtt{v}_i} \mapsto \overline{v_i}\}$ on the rest of the structure

- if the last non-classic component is a module component:

  $\mathtt{v} \triangleright v : \mathsf{Ty}(\mathsf{Funsig}(\hat{v} : M_1)M_2) := m$

  then we simply remove it

- if the last non-classic component is a module or a block component with a manifest specification:

  $\overline{\mathtt{v}} \triangleright \overline{v} : \mathsf{Eq}(\overline{p} : \Psi) := P$

  or if it is implemented by a path or a sequence of paths

  $\overline{\mathtt{v}} \triangleright \overline{v} : \mathsf{Ty}(\Psi) := \overline{p}$

  then we remove it and substitute $\{\overline{v} \mapsto \overline{p}\}$ on the rest of the structure.

Note that all steps decrease the total number of components in the structure and its substructures, and hence the flattening phase is terminating. The resulting structure is of course classic.

Proving the typability preservation of the flattening phase is quite difficult. Because of potential nontrivial subtypings involved in the derivation of types for term paths, the induction hypothesis of the straightforward proof would have to establish the correctness of a signature flattening in an arbitrary environment. Unfortunately the latter does not hold. Instead, we show that subtyping is not essential for term derivations and that flattening of signatures is possible in the absence of subtyping.

Technically, the above proof is done with an auxiliary typing system, called *principal* and denoted $\vdash_{\mathcal{P}}$, which is presented and studied in Chapter 5. It consists of term rules and elimination rules for signatures, specifications and block types. We prove that all term judgments derivable in $\vdash$ can be derived in $\vdash_{\mathcal{P}}$. In this system it becomes easy to prove the correctness of signature flattening by a straightforward induction on the derivation.

The correctness of the remaining steps (removing module and block components involving paths) follows easily from the path substitution lemma.

Note that the goal of the formal development, the term t, does not change during the flattening phase. Indeed, it does not contain names of the non-classic components of the structure and is therefore unaffected by the substitutions. Its proof e can be changed of course.

Chapter 4

Basic metatheory

In this chapter with prove basic metatheoretical facts about our calculus of named modules. Most properties of this section are clones of the corresponding properties in [32]. Unfortunately, we had to repeat the proofs as our typing system is richer than the original one.

4.1 General properties

First we show that a correct environment assigns correct specifications to variables and that derivations are only possible in correct environments.

Lemma 4.1.1. *For every environment $E = \overline{v}_1 : S_1 \dots \overline{v}_n : S_n$, every derivation of $E \vdash \mathtt{ok}$ contains a derivation of $\overline{v}_1 : S_1 \dots \overline{v}_{i-1} : S_{i-1} \vdash \mathtt{ok}$ and a derivation of $\overline{v}_1 : S_1 \dots \overline{v}_{i-1} : S_{i-1} \vdash S_i : \mathtt{spec}$ for every $i = 1 \dots n$.*

Proof. By induction on the derivation of $E \vdash \mathtt{ok}$. The case of the **(Env/Empty)** rule is trivial. If the last rule in the derivation is

$$\textbf{(Env/Insert)} \qquad \frac{E' \vdash \mathtt{ok} \qquad E' \vdash S : \mathtt{spec}}{E'; \overline{v} : S \vdash \mathtt{ok}}$$

then the conclusions of the lemma hold for E' by induction hypothesis. We are left to check $E' \vdash \mathtt{ok}$ and $E' \vdash S : \mathtt{spec}$ which are true as they are the premises of the rule. □

Lemma 4.1.2. *For every environment E and judgment J, every derivation of $E \vdash J$, contains a derivation of $E \vdash \mathtt{ok}$.*

Proof. By induction on the derivation. Let us look at the last rule in the derivation of $E \vdash J$. If J is $\mathtt{ok}$ then the lemma trivially holds. In all other rules there is always at least one premise whose environment is equal to E or extends it. Hence, we can conclude by induction hypothesis (and by Lemma 4.1.1 if the environment is an extension of E). □

The next lemma shows that in our typing system subterms of well-typed expressions are well-typed.

Lemma 4.1.3. *For every judgment $E \vdash \phi : \Psi$, for every position q such that $\phi|_q$ is an expression, there is a type, a sort or a specification Ψ' such that $E; BV(\phi, q) \vdash \phi|_q : \Psi'$.*

Proof. By induction on the derivation of $E \vdash \phi : \Psi$. Let us look at the last rule in this derivation. For most rules the lemma easily holds because all immediate subterms of ϕ appear on the left hand side of the colon symbol in one of the premises of the rule. The rule **(Sig/Form)** needs a bit more work:

$$\frac{E; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n \vdash \mathtt{ok}}{E \vdash \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End} : \mathtt{modtype}}$$

We first use Lemma 4.1.1 to find subderivations $E; \overline{v}_1 : S_1 \ldots \overline{v}_{i-1} : S_{i-1} \vdash S_i : \mathtt{spec}$ for every $i = 1 \ldots n$ and then deduce by induction hypothesis that all subterms of $S_1, \ldots S_n$ have types. □

Lemma 4.1.4. *For all environments E_1, E_2 and E_3, if $E_1; E_2 \vdash \mathtt{ok}$ and $E_1; E_3 \vdash J$ for some judgment J, then $E_1; E_2; E_3 \vdash J$.*

Proof. By induction on the derivation of $E_1; E_3 \vdash J$. Let us look at the last rule in this derivation.

Environment In case of **(Env/Empty)** both E_1 and E_3 are empty and we need to show that $E_2 \vdash \mathtt{ok}$ which is one of the assumptions of the lemma (for $E_1 = \epsilon$).

For **(Env/Insert)**

$$\frac{E \vdash \mathtt{ok} \qquad E \vdash S : \mathtt{spec}}{E; \overline{v} : S \vdash \mathtt{ok}}$$

we have two cases:

1. $E_1 = E; \overline{v} : S$ and $E_3 = \epsilon$. Then $E_1; E_2; E_3 \vdash J$ becomes $E_1; E_2 \vdash \mathtt{ok}$ which is one of the assumptions of the lemma.
2. $E_1 = E'$, $E_3 = E''; \overline{v} : S$ and $E'; E'' = E$. By induction hypothesis applied to the premises of the rule we have $E'; E_2; E'' \vdash \mathtt{ok}$ and $E'; E_2; E'' \vdash S : \mathtt{spec}$. Using the rule **(Env/Insert)** we get $E'; E_2; E'', \overline{v} : S \vdash \mathtt{ok}$ which is the desired conclusion.

For **(Env/Lookup)**

$$\frac{\hat{E}_1; \overline{v} : S; \hat{E}_2 \vdash \mathtt{ok}}{\hat{E}_1; \overline{v} : S; \hat{E}_2 \vdash \overline{v} : S}$$

we have two cases:

1. $E_1 = \hat{E}_1; \overline{v} : S; \hat{E}_2{}'$, $E_3 = \hat{E}_2{}''$ and $\hat{E}_2{}'; \hat{E}_2{}'' = \hat{E}_2$.
   By assumption we know that $\hat{E}_1; \overline{v} : S; \hat{E}_2{}'; E_2 \vdash \mathtt{ok}$. Applying the induction hypothesis to the premise of the rule $\hat{E}_1; \overline{v} : S; \hat{E}_2{}'; \hat{E}_2{}'' \vdash \mathtt{ok}$, we get $\hat{E}_1; \overline{v} : S; \hat{E}_2{}'; E_2; \hat{E}_2{}'' \vdash \mathtt{ok}$ and by the rule **(Env/Lookup)** $\hat{E}_1; \overline{v} : S; \hat{E}_2{}'; E_2; \hat{E}_2{}'' \vdash \overline{v} : S$ which is the desired conclusion.

2. $E_1 = \hat{E_1}'$, $E_3 = \hat{E_1}''; \overline{v} : S; \hat{E_2}$ and $\hat{E_1}'; \hat{E_1}'' = \hat{E_1}$.

   By assumption we know that $\hat{E_1}'; E_2 \vdash \mathtt{ok}$. Applying the induction hypothesis to the premise of the rule $\hat{E_1}'; \hat{E_1}''; \overline{v} : S; \hat{E_2} \vdash \mathtt{ok}$, we get $\hat{E_1}'; E_2; \hat{E_1}''; \overline{v} : S; \hat{E_2} \vdash \mathtt{ok}$ and by the rule **(Env/Lookup)** $\hat{E_1}'; E_2; \hat{E_1}''; \overline{v} : S; \hat{E_2} \vdash \overline{v} : S$ which is the desired conclusion.

Terms Let us consider the **(Term/Prod)** rule

$$\frac{E \vdash t_1 : \mathrm{s}_1 \qquad E; v : \mathsf{Ty}(t_1) \vdash t_2 : \mathrm{s}_2}{E \vdash (v : t_1)t_2 : \mathrm{s}_3} \text{ if } (\mathrm{s}_1, \mathrm{s}_2, \mathrm{s}_3) \in \mathcal{R}$$

where $E = E_1; E_3$. By induction hypothesis we have $E_1; E_2; E_3 \vdash t_1 : \mathrm{s}_1$ and $E_1; E_2; E_3; v : \mathsf{Ty}(t_1) \vdash t_2 : \mathrm{s}_2$. Using the **(Term/Prod)** rule we get $E_1; E_2; E_3 \vdash (v : t_1)t_2 : \mathrm{s}_3$.

Other rules for terms are dealt with similarly.

Modules, Specifications In all rules, the environments of the premises are extensions of the environment of the conclusion. Hence, the lemma holds by induction hypothesis (as above).

Blocks For block rules the situation is similar to term and module rules, but there are additional side conditions in the rules **(IndSpec/Form)**, **(RewSpec/Form)** and **(RewSpec/Rew)**. Fortunately, these side conditions are satisfied in the extended environment by the closure property (C2).

Another rule to be checked is **(RewSpec/Comp)**:

$$\frac{E \vdash \overline{p} : \mathsf{RewSpec}(E^\Gamma, R) \qquad \delta : \hat{E_i} \to E}{E \vdash e_i^L \theta\delta \approx e_i^R \theta\delta} \text{ where } \theta = \{v_j \mapsto p^j\}_{j=1\ldots n}$$

where $E = E_1; E_3$ and $E^\Gamma = v_1 : Ty(t_1) \ldots v_n : Ty(t_n)$. By induction hypothesis we have $E_1; E_2; E_3 \vdash \overline{p} : \mathsf{RewSpec}(E^\Gamma, R)$. In order to apply **(RewSpec/Comp)** and to get $E_1; E_2; E_3 \vdash e_i^L \theta\delta \approx e_i^R \theta\delta$, we have to make sure that $\delta : \hat{E_i} \to E_1; E_2; E_3$. But, we have $\delta : \hat{E_i} \to E$ which means that $E \vdash v\delta : t\delta$ for all $v : t \in \hat{E_i}$. By induction hypothesis, we have also $E_1; E_2; E_3 \vdash v\delta : t\delta$ for all $v : t \in \hat{E_i}$, i.e. $\delta : \hat{E_i} \to E_1; E_2; E_3$.

Convertibility Like for terms, modules and specifications.

□

4.1.1 Substitutivity of paths

Lemma 4.1.5. *For all environments E_1 and E_2, judgment J, specification S and a sequence of paths $\overline{p}$, if $E_1; \overline{v} : S; E_2 \vdash J$ and $E_1 \vdash \overline{p} : S$ then $E_1; E_2\{\overline{v} \mapsto \overline{p}\} \vdash J\{\overline{v} \mapsto \overline{p}\}$.*

Proof. By induction on the derivation of $E_1; \overline{v} : S; E_2 \vdash J$. We inspect the last rule in this derivation. Let us denote the substitution $\{\overline{v} \mapsto \overline{p}\}$ by δ.

Environment The rule **(Env/Empty)** cannot be the last rule of the derivation.

For **(Env/Insert)**:

$$\frac{E \vdash \mathtt{ok} \qquad E \vdash S' : \mathtt{spec}}{E; \overline{v}' : S' \vdash \mathtt{ok}}$$

we distinguish two cases:

1. $\overline{v}' : S' = \overline{v} : S$, $E_1 = E$ and $E_2 = \epsilon$. We have to show that $E \vdash \mathtt{ok}$, which is a premise of the rule.
2. $E_2 = E', \overline{v}' : S'$, and $E = E_1; \overline{v} : S; E'$. We want to show that $E_1; E'\delta; \overline{v}' : S'\delta \vdash \mathtt{ok}$. By induction hypothesis applied to the premises of the rule we get $E_1; E'\delta \vdash \mathtt{ok}$ and $E_1; E'\delta \vdash S'\delta : \mathtt{spec}$. Using **(Env/Insert)** we get the desired conclusion.

For **(Env/Lookup)**

$$\frac{\hat{E}_1; \overline{v}' : S'; \hat{E}_2 \vdash \mathtt{ok}}{\hat{E}_1; \overline{v}' : S'; \hat{E}_2 \vdash \overline{v}' : S'}$$

we have three cases:

1. $\overline{v}' : S' \in E_1$ and the conclusion of the rule is $\hat{E}_1; \overline{v}' : S'; E'_1; \overline{v} : S; E_2 \vdash \overline{v}' : S'$. We want to show that $\hat{E}_1; \overline{v}' : S', E'_1; E_2\delta \vdash \overline{v}'\delta : S'\delta$.
   By induction hypothesis applied to $\hat{E}_1; \overline{v}' : S'; E'_1; \overline{v} : S; E_2 \vdash \mathtt{ok}$ we have $\hat{E}_1; \overline{v}' : S', E'_1; E_2\delta \vdash \mathtt{ok}$. Using **(Env/Lookup)** we get $\hat{E}_1; \overline{v}' : S', E'_1; E_2\delta \vdash \overline{v}' : S'$ which is the desired conclusion since $\overline{v}$ do not occur in S' and $S'\delta = S'$.
2. $\overline{v}' : S' = \overline{v} : S$ and the conclusion of the rule is $E_1; \overline{v} : S; E_2 \vdash \overline{v} : S$. We want to show that $E_1; E_2\delta \vdash \overline{v}\delta : S\delta$, i.e. $E_1; E_2\delta \vdash \overline{p} : S$.
   By assumption we know that $E_1 \vdash \overline{p} : S$. Since $E_1; E_2\delta \vdash \mathtt{ok}$ (by induction hypothesis applied to the premise of the rule) we can use Lemma 4.1.4 to get $E_1; E_2\delta \vdash \overline{p} : S$.
3. $\overline{v}' : S' \in E_2$ and the conclusion of the rule is $E_1; \overline{v} : S; E'_2; \overline{v}' : S'; \hat{E}_2 \vdash \overline{v}' : S'$. We want to show that $E_1; E'_2\delta; \overline{v}' : S'\delta; E_2\delta \vdash \overline{v}' : S'\delta$.
   By induction hypothesis applied to the premise of the rule we have $E_1; E'_2\delta; \overline{v}' : S'\delta; E_2\delta \vdash \mathtt{ok}$ and using **(Env/Lookup)** we get the desired conclusion.

Terms Let us do the proof for the **(Term/App)** rule. Other cases are similar.

$$\frac{E \vdash e : (v' : t_1)t_2 \qquad E \vdash e' : t_1}{E \vdash e\, e' : t_2\{v' \mapsto e'\}}$$

where $E = E_1; \overline{v} : S; E_2$. By induction hypothesis we have $E_1; E_2\delta \vdash e\delta : ((v' : t_1)t_2)\delta$ and $E_1; E_2\delta \vdash e'\delta : t_1\delta$. Using the **(Term/App)** rule we get $E_1; E_2\delta \vdash (e\delta)(e'\delta) :$

$(t_2\delta)\{v' \mapsto e'\delta\}$, which is the desired conclusion since $(t_2\delta)\{v' \mapsto e'\delta\} = (t_2\{v' \mapsto e'\})\delta$ (because $v' \notin \overline{p}$).

Modules The proofs work by a simple use of the induction hypothesis. We also use the fact (in rules **(Sig/Access)** and **(Mod/App)**) that $p'\delta$ is a path.

The only nontrivial case is **(Strengthenning)** where we have to know that $(M/p')\delta = M\delta/p'\delta$ which can be shown by easy induction on the structure of M.

Specifications The proofs are like those for terms, but simpler.

Blocks For blocks rules the situation is similar to term and module rules, but additionally there are some side conditions in **(IndSpec/Form)**, **(RewSpec/Form)** and **(RewSpec/Rew)**. Fortunately they are satisfied for the substituted environment by the closure property (C1).

Another rule where the proof is not straightforward is **(RewSpec/Comp)**:

$$\frac{E \vdash \overline{p}' : \mathsf{RewSpec}(E^\Gamma, R) \qquad \xi : \hat{E}_i \to E}{E \vdash e_i^L\theta\xi \approx e_i^R\theta\xi} \text{ where } \theta = \{v_j \mapsto p'_j\}_{j=1\ldots n}$$

where $E = E_1; \overline{v} : S; E_2$. By induction hypothesis $E_1; E_2\delta \vdash \overline{p}'\delta : \mathsf{RewSpec}(E^\Gamma\delta, R\delta)$. Since $\xi : \hat{E}_i \to E$ means that $E \vdash \hat{v}\xi : t\xi$ for all $\hat{v} : t \in \hat{E}_i$, by induction hypothesis we have $E_1; E_2\delta \vdash \hat{v}\xi\delta : t\xi\delta$ for all $\hat{v} : t \in \hat{E}_i$ and consequently $\xi\delta : \hat{E}_i \to E_1; E_2\delta$. Let us also take $\theta\delta = \{v_j \mapsto p'_j\delta\}_{j=1\ldots n}$ instead of θ. Using **(RewSpec/Comp)** we get $E_1; E_2\delta \vdash e_i^L\delta(\theta\delta)(\xi\delta) \approx e_i^R\delta(\theta\delta)(\xi\delta)$. It is equivalent to the desired conclusion $(E_1; E_2\delta \vdash (e_i^L\delta\theta\xi)\delta \approx (e_i^R\delta\theta\xi)\delta)$ because the domains of θ and ξ are disjoint with the codomain of δ.

Convertibility The proofs are like for term and module rules. The only nontrivial case is **(Conv/Refl)**:

$$\frac{E_1; \overline{v} : S; E_2 \vdash v' : M}{E_1; \overline{v} : S; E_2 \vdash v' \approx v'}$$

If $v' \notin \overline{v}$ then we have $E_1; E_2\delta \vdash v' : M\delta$ by induction hypothesis and $E_1; E_2\delta \vdash v' \approx v'$ by **(Conv/Refl)**.

If $v' \in \overline{v}$ then S is a module specification and $\mathit{length}(\overline{v}) = \mathit{length}(\overline{p}) = 1$. We have to show that $E_1; E_2\delta \vdash p \approx p$. Recall that $E_1 \vdash p : S$ by assumption of the lemma. Let us prove that $E_1 \vdash p \approx p$ by induction on the structure of $p = v''.\mathtt{v}_1 \ldots .\mathtt{v}_n$.

If $n = 0$ then $E_1 \vdash v'' \approx v''$ follows from **(Conv/Refl)**. Suppose now that $p = p'.\mathtt{v}_{n+1}$. By Lemma 4.1.3 we have $E_1 \vdash p' : M'$ for some M' and by induction hypothesis $E_1 \vdash p' \approx p'$. Now, it is sufficient to use **(Conv/Path)** to get $E_1 \vdash p \approx p$.

Going back to the main proof, we can now use Lemma 4.1.4 on $E_1 \vdash p \approx p$ to get $E_1; E_2\delta \vdash p \approx p$ which is the desired conclusion.

$\square$

Lemma 4.1.6. *Let E_1, E_2, E be environments, J a judgment, and δ a substitution assigning paths to variables. If $E_1; E_2 \vdash J$, $E \vdash \mathtt{ok}$ and $E \vdash \overline{v}\delta : S\delta$ for every $\overline{v} : S \in E_1$ then $E; E_2\delta \vdash J\delta$.*

Proof. Let $E_1 = \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n$. By Lemma 4.1.4 we have $E; E_1; E_2 \vdash J$. Now, it is sufficient to eliminate variables from E_1 from left to right, using Lemma 4.1.5 with a substitution which assigns a sequence of paths $\overline{v}_i\delta$ to every $\overline{v}_i$. $\square$

The following lemma is very useful throughout all metatheoretical studies. It says that variables in the environment can be replaced by others, provided the new ones satisfy the original specifications.

Lemma 4.1.7. *Let E_1, E_2 be environments and $S_1 \ldots S_k$ and $S'_1 \ldots S'_m$ specifications. Suppose that*

$$E_1; \overline{v}_1 : S_1 \ldots \overline{v}_k : S_k \vdash \mathtt{ok} \quad \text{and} \quad E_1; \overline{v}'_1 : S'_1 \ldots \overline{v}'_m : S'_m; E_2 \vdash J.$$

If there exists a substitution $\theta : \{\overline{v}'_1 \ldots \overline{v}'_m\} \to \{\overline{v}_1 \ldots \overline{v}_k\}$ such that for all $i = 1 \ldots m$ $E_1; \overline{v}_1 : S_1 \ldots \overline{v}_k : S_k \vdash \overline{v}'_i\theta : S'_i\theta$ then $E_1; \overline{v}_1 : S_1 \ldots \overline{v}_k : S_k; E_2\theta \vdash J\theta$.

Proof. It is a corollary from Lemma 4.1.6. Let $E_v = \overline{v}_1 : S_1 \ldots \overline{v}_k : S_k$, $E'_v = \overline{v}'_1 : S'_1 \ldots \overline{v}'_m : S'_m$, $E = E_1; E_v$ and $E'_1 = E_1; E'_v$. By assumptions of the lemma we have $E'_1; E_2 \vdash J$, $E \vdash \mathtt{ok}$ and by extending θ to be the identity on the domain of E_1, we get $E \vdash \overline{v}\theta : S\theta$ for every $vv : S \in E'_1$. Of course θ is a path substitution and we can use Lemma 4.1.6 to get $E; E_2\theta \vdash J\theta$ which is $E_1; \overline{v}_1 : S_1 \ldots \overline{v}_k : S_k; E_2\theta \vdash J\theta$. $\square$

4.1.2 Module types, subtyping and convertibility

In this subsection we show that our system has expected properties with respect to module types. First, we show that a correct signature contains correct specifications. Then we prove several lemmas about subtyping, convertibility and their relationship. We end with Lemma 4.1.15 stating, among others, that if a module expression has a module type then the latter is correct.

Lemma 4.1.8. *In every derivation of $E \vdash \mathsf{Sig}\ \mathtt{v}_1 \triangleright \overline{v}_1 : S_1 \ldots \mathtt{v}_n \triangleright \overline{v}_n : S_n\ \mathsf{End} : \mathtt{modtype}$ there is a derivation of $E; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n \vdash \mathtt{ok}$ and a derivation of $E; \overline{v}_1 : S_1 \ldots \overline{v}_{i-1} : S_{i-1} \vdash S_i : \mathtt{spec}$ for all $i = 1 \ldots n$.*

Proof. The **(Sig/Form)** rule is the only one that can be used to derive the judgment $E \vdash \mathsf{Sig}\ \mathtt{v}_1 \triangleright \overline{v}_1 : S_1 \ldots \mathtt{v}_n \triangleright \overline{v}_n : S_n\ \mathsf{End} : \mathtt{modtype}$. Its premises give the first conclusion of our lemma. The second follows from Lemma 4.1.1. $\square$

Lemma 4.1.9. *For every environment E and module types M_1, M_2 and M_3, if $E \vdash M_1 <: M_2$ and $E \vdash M_2 <: M_3$ then $E \vdash M_1 <: M_3$.*

Proof. It is easy to see that M_1, M_2 and M_3 are either all functor types or all signatures. We obtain the result by structural induction on the concerned module types.

1. Let $M_1 = \mathsf{Funsig}(v : M_1')M_1''$, $M_2 = \mathsf{Funsig}(v : M_2')M_2''$ and $M_3 = \mathsf{Funsig}(v : M_3')M_3''$. The last rules in the derivations of $E \vdash M_1 <: M_2$ and $E \vdash M_2 <: M_3$ have to be **(Funsig/Sub)**. The premises of these rules give us: $E \vdash M_2' <: M_1'$, $E, v : \mathsf{Ty}(M_2') \vdash M_1'' <: M_2''$ and $E \vdash M_3' <: M_2'$, $E, v : \mathsf{Ty}(M_3') \vdash M_2'' <: M_3''$. By induction hypothesis we have $E \vdash M_3' <: M_1'$. If we manage to prove $E, v : \mathsf{Ty}(M_3') \vdash M_1'' <: M_3''$, we can apply **(Funsig/Sub)** to get $E \vdash \mathsf{Funsig}(v : M_1')M_1'' <: \mathsf{Funsig}(v : M_3')M_3''$.

   In order to get $E, v : \mathsf{Ty}(M_3') \vdash M_1'' <: M_3''$ we apply induction hypothesis to $E, v : \mathsf{Ty}(M_3') \vdash M_2'' <: M_3''$ and $E, v : \mathsf{Ty}(M_3') \vdash M_1'' <: M_2''$ that we can prove by Lemma 4.1.7. Note that we have $E; v : \mathsf{Ty}(M_3') \vdash \mathtt{ok}$ by Lemma 4.1.2 and $E, v : \mathsf{Ty}(M_2') \vdash M_1'' <: M_2''$ is one of the premises listed above. If we had $E, v : \mathsf{Ty}(M_3') \vdash v : \mathsf{Ty}(M_2')$ we could use Lemma 4.1.7 and get $E, v : \mathsf{Ty}(M_3') \vdash M_1'' <: M_3''$.

   We are left to show $E, v : \mathsf{Ty}(M_3') \vdash v : \mathsf{Ty}(M_2')$. Starting from $E; v : \mathsf{Ty}(M_3') \vdash v : \mathsf{Ty}(M_3')$, which is true as $E, v : \mathsf{Ty}(M_3') \vdash \mathtt{ok}$, we can get $E, v : \mathsf{Ty}(M_3') \vdash v : M_3'$ by **(Ty/Type)**. Then we apply **(Subsumption)** with $E, v : \mathsf{Ty}(M_3') \vdash M_3' <: M_2'$ (which is true since we have $E \vdash M_3' <: M_2'$ and $E, v : \mathsf{Ty}(M_3') \vdash \mathtt{ok}$) and get $E, v : \mathsf{Ty}(M_3') \vdash v : M_2'$. Finally, by **(Ty/Sat)**, we obtain $E, v : \mathsf{Ty}(M_3') \vdash v : \mathsf{Ty}(M_2')$. The use of **(Ty/Sat)** is correct because we have $E, v : \mathsf{Ty}(M_2') \vdash \mathtt{ok}$ and by Lemma 4.1.1 $E \vdash \mathsf{Ty}(M_2') : \mathtt{spec}$.

2. Let $M_1 = \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$, $M_2 = \mathsf{Sig}\ \overline{\mathtt{v}}_1' \triangleright \overline{v}_1' : S_1' \ldots \overline{\mathtt{v}}_{n'}' \triangleright \overline{v}_{n'}' : S_{n'}'\ \mathsf{End}$ and $M_3 = \mathsf{Sig}\ \overline{\mathtt{v}}_1'' \triangleright \overline{v}_1'' : S_1'' \ldots \overline{\mathtt{v}}_{n''}'' \triangleright \overline{v}_{n''}'' : S_{n''}''\ \mathsf{End}$. The last rules in the derivations of $E \vdash M_1 <: M_2$ and $E \vdash M_2 <: M_3$ have to be **(Sig/Sub)**. The premises of these rules give us: $E; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n \vdash \overline{v}_k' : S_k'$ for $k = 1 \ldots n'$ and $E; \overline{v}_1' : S_1' \ldots \overline{v}_n' : S_n' \vdash \overline{v}_k'' : S_k''$ for $k = 1 \ldots n''$. We want to show that $E; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n \vdash \overline{v}_k'' : S_k''$ for $k = 1 \ldots n''$.

   To this end for every $k = 1 \ldots n''$ we can use Lemma 4.1.7. We have $E; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n \vdash \mathtt{ok}$, $E; \overline{v}_1' : S_1' \ldots \overline{v}_{n'}' : S_{n'}' \vdash \overline{v}_k'' : S_k''$ and $E; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n \vdash \overline{v}_i' : S_i'$ for $i = 1 \ldots n'$. Consequently, $E; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n \vdash \overline{v}_k'' : S_k''$.

$\square$

The following lemma shows that convertible module types are subtypes of each other. Of course we have to consider specifications in the same time.

Lemma 4.1.10. *For every environment E, module types M_1, M_2 and specifications S_1, S_2:*

- *if $E \vdash M_1 \approx M_2$ then $E \vdash M_1 <: M_2$ and $E \vdash M_2 <: M_1$*
- *if $E \vdash S_1 \approx S_2$ then $E; \overline{v} : S_1 \vdash \overline{v} : S_2$ and $E; \overline{v} : S_2 \vdash \overline{v} : S_1$.*

Proof. By induction on the derivation of $E \vdash M_1 \approx M_2$ and $E \vdash S_1 \approx S_2$.

1. Let $E \vdash M_1 \approx M_2$. There are two possibilities for M_1 and M_2. Either they are both functor types or signatures:

(a) $M_1 = \mathsf{Funsig}(v : M_1')M_1''$, $M_2 = \mathsf{Funsig}(v : M_2')M_2''$. Since $E \vdash M_1 \approx M_2$ can be obtained only by **(Conv/Funsig)** we know that $E \vdash \mathsf{Funsig}(v : M_1')M_1'' : \mathtt{modtype}$, $E \vdash M_1' \approx M_2'$, $E; v : \mathsf{Ty}(M_1') \vdash M_1'' \approx M_2''$ and $E \vdash \mathsf{Funsig}(v : M_2')M_2'' : \mathtt{modtype}$. By induction hypothesis we get

$$E \vdash M_1' <: M_2' \tag{4.1}$$
$$E \vdash M_2' <: M_1' \tag{4.2}$$
$$E; v : \mathsf{Ty}(M_1') \vdash M_1'' <: M_2'' \tag{4.3}$$
$$E; v : \mathsf{Ty}(M_1') \vdash M_2'' <: M_1'' \tag{4.4}$$

From $E; v : \mathsf{Ty}(M_1') \vdash M_1'' <: M_2''$ and $E; v : \mathsf{Ty}(M_2') \vdash v : \mathsf{Ty}(M_1')$ (which is true by induction hypothesis) we get $E; v : \mathsf{Ty}(M_2') \vdash M_1'' <: M_2''$ by Lemma 4.1.7. Applying **(Funsig/Sub)** to (4.1) and (4.4) we get $E \vdash \mathsf{Funsig}(v : M_2')M_2'' <: \mathsf{Funsig}(v : M_1')M_1''$ and applying **(Funsig/Sub)** to (4.2) and $E; v : \mathsf{Ty}(M_2') \vdash M_1'' <: M_2''$ we get $E \vdash \mathsf{Funsig}(v : M_1')M_1'' <: \mathsf{Funsig}(v : M_2')M_2''$.

(b) $M_1 = \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$, $M_2 = \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1' \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n'\ \mathsf{End}$. Since $E \vdash M_1 \approx M_2$ can be obtained only by **(Conv/Sig)** we know that $E \vdash M_1 : \mathtt{modtype}$, $E \vdash M_2 : \mathtt{modtype}$ and $E; \overline{v}_1 : S_1 \ldots \overline{v}_{i-1} : S_{i-1} \vdash S_i \approx S_i'$ for $i = 1 \ldots n$. By induction hypothesis $E; \overline{v}_1 : S_1 \ldots \overline{v}_{i-1} : S_{i-1}; \overline{v}_i : S_i \vdash \overline{v}_i : S_i'$ for $i = 1 \ldots n$. Since $E; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n \vdash \mathtt{ok}$ we can use Lemma 4.1.4 to get $E; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n \vdash \overline{v}_i : S_i'$ for $i = 1 \ldots n$. By **(Sig/Sub)** we obtain $E \vdash \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End} <: \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1' \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n'\ \mathsf{End}$.

To get $E \vdash \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1' \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n'\ \mathsf{End} <: \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$ we need judgments $E; \overline{v}_1 : S_1' \ldots \overline{v}_{i-1} : S_{i-1}'; \overline{v}_i : S_i' \vdash \overline{v}_i : S_i$ for $i = 1 \ldots n$. From $E; \overline{v}_1 : S_1 \ldots \overline{v}_{i-1} : S_{i-1} \vdash S_i \approx S_i'$ for $i = 1 \ldots n$ we get by induction hypothesis $E; \overline{v}_1 : S_1 \ldots \overline{v}_{i-1} : S_{i-1}; \overline{v}_i : S_i' \vdash \overline{v}_i : S_i$ for $i = 1 \ldots n$. For $i = 1$ we have $E; \overline{v}_1 : S_1' \vdash \overline{v}_1 : S_1$. For i=2 we have $E, \overline{v}_1 : S_1, \overline{v}_2 : S_2' \vdash \overline{v}_2 : S_2$ and since $E; \overline{v}_1 : S_1', \overline{v}_2 : S_2' \vdash \overline{v}_1 : S_1$ (it follows from $E; \overline{v}_1 : S_1' \vdash \overline{v}_1 : S_1$ by Lemma 4.1.4) we can use Lemma 4.1.7 and get $E, \overline{v}_1 : S_1', \overline{v}_2 : S_2' \vdash \overline{v}_2 : S_2$. We continue like this till $i = n$ and then apply **(Sig/Sub)**.

2. Let $E \vdash S_1 \approx S_2$. There are two possibilities for S_1 and S_2, they are both manifest or abstract specifications. Let us do the proof for manifest ones: $S_1 = \mathsf{Eq}(\phi_1 : \Psi_1)$, $S_2 = \mathsf{Eq}(\phi_2 : \Psi_2)$. Since $E \vdash S_1 \approx S_2$ can be obtained only by **(Conv/Eq)** we know that $E \vdash S_1 : \mathtt{spec}$, $E \vdash \phi_1 \approx \phi_2$, $E \vdash \Psi_1 \approx \Psi_2$, $E \vdash S_2 : \mathtt{spec}$. There are still three possibilities: this can be a module, a term or a block specification.

   (a) Module specification, $S_1 = \mathsf{Eq}(p_1 : M_1)$, $S_2 = \mathsf{Eq}(p_2 : M_2)$. By induction hypothesis applied to $E \vdash M_1 \approx M_2$ we get $E \vdash M_1 <: M_2$ and $E \vdash M_2 <: M_1$. Now it is easy to get the desired conclusion. Starting from $E; v : S_1 \vdash \overline{v} : S_1$, by **(Eq/Type)** we get $E; v : S_1 \vdash v : M_1$ and by **(Subsumption)** we get $E; v : S_1 \vdash v : M_2$. Applying **(Eq/Comp)** to $E; v : S_1 \vdash v : S_1$ gives $E; v : S_1 \vdash v \approx p_1$ and since $E; v : S_1 \vdash p_1 \approx p_2$ by Lemma 4.1.4, we get $E; v : S_1 \vdash v \approx p_2$.

Now, by **(Eq/Sat)** we get $E; v : S_1 \vdash v : \mathsf{Eq}(p_2 : M_2)$, i.e. $E; v : S_1 \vdash v : S_2$. Similarily using **(Conv/Symm)**, we can prove $E; v : S_2 \vdash v : S_1$.

(b) Term or block specification. In this case we use conversion **(Term/Conv)** or **(Block/Conv)** instead of **(Subsumption)**. Starting from $E; \overline{v} : S_1 \vdash \overline{v} : S_1$, by **(Eq/Type)** we get $E; \overline{v} : S_1 \vdash \overline{v} : \Psi_1$, by conversion we get $E; \overline{v} : S_1 \vdash \overline{v} : \Psi_2$, and by **(Eq/Sat)** we finally obtain $E; \overline{v} : S_1 \vdash \overline{v} : \mathsf{Eq}(\phi_2 : \Psi_2)$. □

Lemma 4.1.11. *For every environment E and specifications S_1, S_2, if $E \vdash \phi : S_1$ and $E \vdash S_1 \approx S_2$ then $E \vdash \phi : S_2$.*

Proof. The judgment $E \vdash S_1 \approx S_2$ was either proved by **(Conv/Ty)** or **(Conv/Eq)**. By inverting these rules we get:

1. $S_1 = \mathsf{Ty}(\Psi_1)$, $S_2 = \mathsf{Ty}(\Psi_2)$, $E \vdash \Psi_1 \approx \Psi_2$, $E \vdash \mathsf{Ty}(\Psi_1) : \mathtt{spec}$ and $E \vdash \mathsf{Ty}(\Psi_2) : \mathtt{spec}$.

   (a) if Ψ_1, Ψ_2 are terms or block types then $E \vdash \phi : \mathsf{Ty}(\Psi_1)$ implies $E \vdash \phi : \Psi_1$ and $E \vdash \mathsf{Ty}(\Psi_2) : \mathtt{spec}$ implies $E \vdash \Psi_2 : \mathrm{s}$, if Ψ_2 is a term, and $E \vdash \Psi_2 : \mathtt{blocktype}$ otherwise. By **(Term/Conv)** (or **(Block/Conv)**) we get $E \vdash \phi : \Psi_2$ and by **(Ty/Sat)** $E \vdash \phi : \mathsf{Ty}(\Psi_2)$.

   (b) if Ψ_1, Ψ_2 are module types then $E \vdash \phi : \mathsf{Ty}(\Psi_1)$ implies $E \vdash \phi : \Psi_1$. By Lemma 4.1.10 we have $E \vdash \Psi_1 <: \Psi_2$ and by **(Subsumption)** $E \vdash \phi : \Psi_2$. Finally, by **(Ty/Sat)**, we get $E \vdash \phi : \mathsf{Ty}(\Psi_2)$.

2. $S_1 = \mathsf{Eq}(\phi_1 : \Psi_1)$, $S_2 = \mathsf{Ty}(\phi_2 : \Phi_2)$, $E \vdash \phi_1 \approx \phi_2$, $E \vdash \Psi_1 \approx \Psi_2$, $E \vdash \mathsf{Eq}(\phi_1 : \Psi_1) : \mathtt{spec}$ and $E \vdash \mathsf{Eq}(\phi_2 : \Psi_2) : \mathtt{spec}$.

   (a) if Ψ_1, Ψ_2 are terms or block types then $E \vdash \phi : \mathsf{Eq}(\phi_1 : \Psi_1)$ implies $E \vdash \phi : \Psi_1$ and $E \vdash \mathsf{Eq}(\phi_2 : \Psi_2) : \mathtt{spec}$ implies $E \vdash \Psi_2 : \mathrm{s}$ if Ψ_2 is a term, and $E \vdash \Psi_2 : \mathtt{blocktype}$ otherwise. By **(Term/Conv)** (or **(Block/Conv)**) we get $E \vdash \phi : \Psi_2$. Moreover, we have $E \vdash \phi \approx \phi_1$ by **(Eq/Comp)** applied to $E \vdash \phi : \mathsf{Eq}(\phi_1 : \Psi_1)$, $E \vdash \phi_1 \approx \phi_2$ by assumption and $E \vdash \phi \approx \phi_2$ by **(Conv/Trans)**. By **(Eq/Sat)** we get the desired conclusion $E \vdash \phi : \mathsf{Eq}(\phi_2 : \Psi_2)$.

   (b) if Ψ_1, Ψ_2 are module types then $E \vdash \phi : \mathsf{Eq}(\phi_1 : \Psi_1)$ implies $E \vdash \phi : \Psi_1$. By Lemma 4.1.10 we have $E \vdash \Psi_1 <: \Psi_2$ and by **(Subsumption)** $E \vdash \phi : \Psi_2$. Like above we prove that $E \vdash \phi \approx \phi_2$ and finally, by **(Eq/Sat)**, we get $E \vdash \phi : \mathsf{Eq}(\phi_2 : \Psi_2)$. □

Lemma 4.1.12. *For every environment E and Ψ_1, Ψ_2, Ψ_3 that are all module types, specifications or block types*

- *if $E \vdash \Psi : \Xi$ then $E \vdash \Psi \approx \Psi$ (reflexivity)*
- *if $E \vdash \Psi_1 \approx \Psi_2$ then $E \vdash \Psi_2 \approx \Psi_1$ (symmetry)*
- *if $E \vdash \Psi_1 \approx \Psi_2$ and $E \vdash \Psi_2 \approx \Psi_3$ then $E \vdash \Psi_1 \approx \Psi_3$ (transitivity).*

Proof. Let us do the proofs first for module types, then for specifications and finally for block types. The proofs are done by structural induction.

1. Module types

   Reflexivity There are two cases: functor types and signatures.

   (a) If $M = \mathsf{Funsig}(v : M')M''$ then $E \vdash M' : \mathtt{modtype}$ and $E; v : \mathsf{Ty}(M') \vdash M'' : \mathtt{modtype}$, since $E \vdash M : \mathtt{modtype}$ can be obtained only by **(Funsig/Form)**. By induction hypothesis we have $E \vdash M' \approx M'$ and $E; v : \mathsf{Ty}(M') \vdash M'' \approx M''$. By **(Conv/Sig)** $E \vdash \mathsf{Funsig}(v : M')M'' \approx \mathsf{Funsig}(v : M')M''$.

   (b) If $M = \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$ and $E \vdash M : \mathtt{modtype}$ then $E; \overline{v}_1 : S_1 \ldots \overline{v}_{i-1} : S_{i-1} \vdash S_i : \mathtt{spec}$ for every $i = 1 \ldots n$. By induction hypothesis we have $E; \overline{v}_1 : S_1 \ldots \overline{v}_{i-1} : S_{i-1} \vdash S_i \approx S_i$ for every $i = 1 \ldots n$ and by **(Conv/Sig)** we get the desired conclusion $E \vdash M \approx M$.

   Symmetry There two cases: functor types and signatures.

   (a) If $M_1 = \mathsf{Funsig}(v : M_1')M_1''$ then $M_2 = \mathsf{Funsig}(v : M_2')M_2''$ for some M_2', M_2'' and since their equality can be shown only by **(Conv/Funsig)**, we know that $E \vdash M_1' \approx M_2'$ and $E; v : \mathsf{Ty}(M_1') \vdash M_1'' \approx M_2''$. By induction hypothesis we have $E \vdash M_2' \approx M_1'$. By Lemma 4.1.10 we have $E; v : \mathsf{Ty}(M_2') \vdash v : \mathsf{Ty}(M_1')$ and by Lemma 4.1.7 applied to $E; v : \mathsf{Ty}(M_1') \vdash M_1'' \approx M_2''$ we get $E; v : \mathsf{Ty}(M_2') \vdash M_1'' \approx M_2''$. By induction hypothesis we have $E; v : \mathsf{Ty}(M_2') \vdash M_2'' \approx M_1''$ and by **(Conv/Funsig)** $E \vdash \mathsf{Funsig}(v : M_2')M_2'' \approx \mathsf{Funsig}(v : M_1')M_1''$.

   (b) If $M_1 = \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$ and $E \vdash M_1 \approx M_2$ then $M_2 = \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1' \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n'\ \mathsf{End}$ for some $S_1', \ldots, S_n'$. Since $E \vdash M_1 \approx M_2$ we know that $E \vdash M_1 : \mathtt{modtype}$, $E \vdash M_2 : \mathtt{modtype}$ and $E; \overline{v}_1 : S_1 \ldots \overline{v}_{i-1} : S_{i-1} \vdash S_i \approx S_i'$ for every $i = 1 \ldots n$. Let us show that $E; \overline{v}_1 : S_1' \ldots \overline{v}_{i-1} : S_{i-1}' \vdash S_i \approx S_i'$ for every $i = 1 \ldots n$.
   We have $E \vdash S_1 \approx S_1'$ and by induction hypothesis $E \vdash S_1' \approx S_1$. By Lemma 4.1.11 $E; \overline{v}_1 : S_1' \vdash \overline{v}_1 : S_1$. We also have $E; \overline{v}_1 : S_1 \vdash S_2 \approx S_2'$. If we apply Lemma 4.1.7 then we get $E; \overline{v}_1 : S_1' \vdash S_2 \approx S_2'$ and by induction hypothesis $E; \overline{v}_1 : S_1' \vdash S_2' \approx S_2$. Similarly, we can convert $E; \overline{v}_1 : S_1; \overline{v}_2 : S_2 \vdash S_3 \approx S_3'$ into $E; \overline{v}_1 : S_1'; \overline{v}_2 : S_2' \vdash S_3 \approx S_3'$ (by Lemma 4.1.7, $E; \overline{v}_1 : S_1'; \overline{v}_2 : S_2' \vdash \overline{v}_2 : S_2$, which is true by Lemma 4.1.11, and $E; \overline{v}_1 : S_1'; \overline{v}_2 : S_2' \vdash \overline{v}_1 : S_1$, which is true by Lemma 4.1.4). Then, by induction hypothesis, we get $E; \overline{v}_1 : S_1'; \overline{v}_2 : S_2' \vdash S_3' \approx S_3$. After continuing like that to $i = n$ we can apply **(Conv/Sig)** and obtain $E \vdash \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1' \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n'\ \mathsf{End} \approx \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$.

   Transitivity Like above there are two cases: functor types and signatures.

   (a) Let $M_1 = \mathsf{Funsig}(v : M_1')M_1''$, $M_2 = \mathsf{Funsig}(v : M_2')M_2''$ and $M_3 = \mathsf{Funsig}(v : M_3')M_3''$. From $E \vdash M_1 \approx M_2$ and $E \vdash M_2 \approx M_3$ we know that $E \vdash M_1' \approx M_2'$, $E \vdash M_2' \approx M_3'$, $E; v : \mathsf{Ty}(M_1') \vdash M_1'' \approx M_2''$ and $E; v : \mathsf{Ty}(M_2') \vdash$

$M_2'' \approx M_3''$. By induction hypothesis we have $E \vdash M_1' \approx M_3'$. Like in the symmetry case, by Lemmas 4.1.7 and 4.1.10 we get $E; v : \mathsf{Ty}(M_1') \vdash M_2'' \approx M_3''$ and by induction hypothesis $E; v : \mathsf{Ty}(M_1') \vdash M_1'' \approx M_3''$. Finally, by **(Conv/Funsig)** it follows that $E \vdash \mathsf{Funsig}(v : M_1')M_1'' \approx \mathsf{Funsig}(v : M_3')M_3''$.

(b) Let $M_1 = \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$, $M_2 = \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1' \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n'\ \mathsf{End}$ and $M_3 = \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1'' \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n''\ \mathsf{End}$. From $E \vdash M_1 \approx M_2$ and $E \vdash M_2 \approx M_3$ we know that $E; \overline{v}_1 : S_1 \ldots \overline{v}_{i-1} : S_{i-1} \vdash S_i \approx S_i'$ and $E; \overline{v}_1 : S_1' \ldots \overline{v}_{i-1} : S_{i-1}' \vdash S_i' \approx S_i''$ for every $i = 1 \ldots n$.
Like in the symmetry case we can convert $E; \overline{v}_1 : S_1' \ldots \overline{v}_{i-1} : S_{i-1}' \vdash S_i' \approx S_i''$ into $E; \overline{v}_1 : S_1 \ldots \overline{v}_{i-1} : S_{i-1} \vdash S_i' \approx S_i''$ for $i = 1 \ldots n$. Then by induction hypothesis we have $E; \overline{v}_1 : S_1 \ldots \overline{v}_{i-1} : S_{i-1} \vdash S_i \approx S_i''$ and by **(Conv/Sig)** $E \vdash \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End} \approx \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1'' \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n''\ \mathsf{End}$.

2. Specifications

Reflexivity Let us do the proof for manifest specifications, for abstract ones the proof is similar. Let $S = \mathsf{Eq}(\phi : \Psi)$.

Since $E \vdash S : \mathtt{spec}$ can be obtained only by **(Eq/Form)** we know that $E \vdash \phi : \Psi$ and $E \vdash \Psi : \Xi$. By induction hypothesis $E \vdash \Psi \approx \Psi$. The expression ϕ is a term, a path or a sequence of paths. If it is a term then $E \vdash \phi \approx \phi$ by **(Conv/Beta)**. If it is a path $p = v.\mathtt{v}_1 \ldots v_n$ then we have $E \vdash v \approx v$ from reflexivity for module variables. After n applications of **(Conv/Path)** we get $E \vdash p \approx p$. For sequences of paths the reasonning is similar.

Summarizing, we have $E \vdash \Psi \approx \Psi$ and $E \vdash \phi \approx \phi$. Hence, by **(Conv/Eq)**, we get $E \vdash \mathsf{Eq}(\phi : \Psi) \approx \mathsf{Eq}(\phi : \Psi)$.

Symmetry Let us do the proof for manifest specifications, for abstract ones the proof is similar. Let $S_1 = \mathsf{Eq}(\phi_1 : \Psi_1)$, $S_2 = \mathsf{Eq}(\phi_2 : \Psi_2)$.

Since $E \vdash S_1 \approx S_2$ can result only from the **(Conv/Eq)** rule, we know that $E \vdash S_1 : \mathtt{spec}$, $E \vdash S_2 : \mathtt{spec}$, $E \vdash \phi_1 \approx \phi_2$ and $E \vdash \Psi_1 \approx \Psi_2$. By induction hypothesis $E \vdash \phi_2 \approx \phi_1$ and $E \vdash \Psi_2 \approx \Psi_1$ and by **(Conv/Eq)** $E \vdash \mathsf{Eq}(\phi_2 : \Psi_2) \approx \mathsf{Eq}(\phi_1 : \Psi_1)$.

Transitivity Like above let us do the proof only for manifest specifications $S_1 = \mathsf{Eq}(\phi_1 : \Psi_1)$, $S_2 = \mathsf{Eq}(\phi_2 : \Psi_2)$, $S_3 = \mathsf{Eq}(\phi_3 : \Psi_3)$. From $E \vdash S_1 \approx S_2$ and $E \vdash S_2 \approx S_3$ it follows that $E \vdash \phi_1 \approx \phi_2$, $E \vdash \phi_2 \approx \phi_3$, $E \vdash \Psi_1 \approx \Psi_2$ and $E \vdash \Psi_2 \approx \Psi_3$. By induction hypothesis we have $E \vdash \phi_1 \approx \phi_3$ and $E \vdash \Psi_1 \approx \Psi_3$ and by **(Conv/Eq)** $E \vdash \mathsf{Eq}(\phi_1 : \Psi_1) \approx \mathsf{Eq}(\phi_3 : \Psi_3)$.

3. Block types. Reflexivity, symmetry and transitivity for block types hold because these properties are true for terms, and term judgments are the only ones appearing in the premises of formation rules for block types. In the proofs of symmetry and transitivity, we need to convert environments (change types to convertible ones) like in the symmetry and transitivity cases for module types.

$\square$

Lemma 4.1.13. *Let E be an environment and ϕ an expression which is neither a module nor a block. Let $v_1 \dots v_n$ be free variables of ϕ and δ_1, δ_2 substitutions. If $E \vdash v_i\delta_1 \approx v_i\delta_2$ for every $i = 1 \dots n$ and $E \vdash \phi\delta_1 : \Psi_1$, $E \vdash \phi\delta_2 : \Psi_2$ for some Ψ_1, Ψ_2 then $E \vdash \phi\delta_1 \approx \phi\delta_2$.*

Proof. By induction on the structure of ϕ. Let us do the proof for ϕ being a variable, an abstraction, a signature, a functor type and an $\mathsf{IndSpec}()$. For other expressions proofs are similar.

If ϕ is a variable then $\phi = v_i$ and $E \vdash v_i\delta_1 \approx v_i\delta_2$ holds by assumption about substitutions δ_1 and δ_2.

If $\phi = [v : t]e$ then $E \vdash ([v : t]e)\delta_1 : \Psi_1$ and $E \vdash ([v : t]e)\delta_2 : \Psi_2$ imply, by Lemma 4.1.3, that $E \vdash t\delta_1 : \Psi_1'$, $E; v : \mathsf{Ty}(t\delta_1) \vdash e\delta_1 : \Psi_1''$, $E \vdash t\delta_2 : \Psi_2'$ and $E; v : \mathsf{Ty}(t\delta_2) \vdash e\delta_2 : \Psi_1''$ for some Ψ_1', Ψ_1'', Ψ_2' and Ψ_2''. By induction hypothesis we know that $E \vdash t\delta_1 \approx t\delta_2$. Using Lemma 4.1.7 on $E; v : \mathsf{Ty}(t\delta_1) \vdash \mathtt{ok}$, $E; v : \mathsf{Ty}(t\delta_2) \vdash e\delta_2 : \Psi_1''$ and $E; v : \mathsf{Ty}(t\delta_1) \vdash v : \mathsf{Ty}(t\delta_2)$ we get $E; v : \mathsf{Ty}(t\delta_1) \vdash e\delta_2 : \Psi_1''$. Note that $E; v : \mathsf{Ty}(t\delta_1) \vdash v : \mathsf{Ty}(t\delta_2)$ is true by Lemma 4.1.10 and $E \vdash \mathsf{Ty}(t\delta_1) \approx \mathsf{Ty}(t\delta_2)$, which follows from $E \vdash t\delta_1 \approx t\delta_2$. Now, by induction hypothesis $E; v : \mathsf{Ty}(t\delta_1) \vdash e\delta_1 \approx e\delta_2$ and it is sufficient to use **(Conv/Abs)** to obtain $E \vdash ([v : t]e)\delta_1 \approx ([v : t]e)\delta_2$.

If $\phi = \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \dots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$ then by Lemma 4.1.8 we have $E; \overline{v}_1 : S_1\delta_1 \dots \overline{v}_{i-1} : S_{i-1}\delta_1 \vdash S_i\delta_1 : \mathtt{spec}$, $E; \overline{v}_1 : S_1\delta_2 \dots \overline{v}_{i-1} : S_{i-1}\delta_2 \vdash S_i\delta_2 : \mathtt{spec}$ for $i = 1 \dots n$. In order to get $E; \overline{v}_1 : S_1\delta_1 \dots \overline{v}_{i-1} : S_{i-1}\delta_1 \vdash S_i\delta_2 : \mathtt{spec}$ for $i = 1 \dots n$ we will interleave the use of induction hypothesis and Lemma 4.1.11. First, by induction hypothesis $E \vdash S_1\delta_1 \approx S_1\delta_2$. Using this equality, by Lemma 4.1.7 and 4.1.11 we get $E; \overline{v}_1 : S_1\delta_1 \vdash S_2\delta_2 : \mathtt{spec}$. By induction hypothesis $E; \overline{v}_1 : S_1\delta_1 \vdash S_2\delta_1 \approx S_2\delta_2$. Using this second equality we get $E; \overline{v}_1 : S_1\delta_1, \overline{v}_2 : S_2\delta_1 \vdash S_3\delta_2 : \mathtt{spec}$. After continuing like this till $i = n$, we can use **(Conv/Sig)** to obtain $E \vdash (\mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \dots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End})\delta_1 \approx (\mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \dots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End})\delta_2$.

If $\phi = \mathsf{Funsig}(v : M')M''$ then the proof is similar to the abstraction case, but we need to use Lemma 4.1.10 in order to convert types with **(Subsumption)** (instead of **(Term/Conv)** which does not work for modules).

If $\phi = \mathsf{IndSpec}(E^I := E^C)$ with $E^I = v_1^I : \mathsf{Ty}(t_1) \dots v_k^I : \mathsf{Ty}(t_k)$, $E^C = v_1^C : \mathsf{Ty}(\hat{t}_1) \dots v_k^C : \mathsf{Ty}(\hat{t}_n)$ then by Lemma 4.1.3 we have $E \vdash t_j\delta_1 : \Psi_j^1$, $E \vdash t_j\delta_2 : \Psi_j^2$ for $j = 1 \dots k$ and $E; E^I\delta_1 \vdash \hat{t}_i : \Psi_i^{1\prime}$, $E; E^I\delta_2 \vdash \hat{t}_i : \Psi_i^{2\prime}$ for $i = 1 \dots n$ for some Ψ_j^1, Ψ_j^2, $\Psi_i^{1\prime}$ and $\Psi_i^{2\prime}$. By induction hypothesis $E \vdash t_j\delta_1 \approx t_j\delta_2$ for $j = 1 \dots k$. In order to get $E; E^I\delta_1 \vdash \hat{t}_i : \Psi_i^{2\prime}$ from $E; E^I\delta_2 \vdash \hat{t}_i : \Psi_i^{2\prime}$ we have to proceed like in the abstraction case and use conversion for all variables of E^I. Afterwards, we can use induction hypothesis, get $E; E^I\delta_1 \vdash \hat{t}_i\delta_1 \approx \hat{t}_i\delta_2$ for $i = 1 \dots n$ and apply **(Conv/IndSpec)** to obtain $E \vdash \mathsf{IndSpec}(E^I := E^C)\delta_2 \approx \mathsf{IndSpec}(E^I := E^C)\delta_2$. $\square$

Apart from subtyping, another operation performed on module types is strengthenning. It is not difficult to understand but quite cumbersome for formal proof of its correctness.

Lemma 4.1.14. *For every environment E, module type M, path p and for every specification S and path sequence $\overline{p}$:*

1. *if* $E \vdash M : \texttt{modtype}$ *and* $E \vdash p : M$ *then* $E \vdash M/p : \texttt{modtype}$ *and* $E \vdash M/p <: M$. *Moreover if* $M = \mathsf{Sig}\ \mathtt{\overline{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \mathtt{\overline{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$ *then for every* $i = 1 \ldots n$

   (a) $E \vdash S_i\{\overline{v}_j \mapsto \overline{p.\mathtt{v}_j}\} : \texttt{spec}$

   (b) $E_{i-1/} \vdash \overline{v}_{i-1} : S_{i-1}$

   (c) $E_{i-1/} \vdash S_i : \texttt{spec}$

   (d) $E_{i-1/} \vdash v^j_{i-1} \approx p.\mathtt{v}^j_{i-1}$

   (e) $E_{i-1/} \vdash S_i \approx S_i\{\overline{v}_j \mapsto \overline{p.\mathtt{v}_j}\}$

   (f) $E_{i-1/} \vdash \overline{p.\mathtt{v}_i} : S_i$

   (g) $E_{i-1/} \vdash \overline{p.\mathtt{v}_i} : S_i/\overline{p.\mathtt{v}_i}$

   where $E_{i/} = E, \overline{v}_1 : S_1/\overline{p.\mathtt{v}_1} \ldots \overline{v}_i : S_i/\overline{p.\mathtt{v}_i}$.

2. *if* $E \vdash S : \texttt{spec}$ *and* $E \vdash \overline{p} : S$ *then* $E \vdash S/\overline{p} : \texttt{spec}$, $E \vdash \overline{p} : S/\overline{p}$, $E; \overline{v} : S/\overline{p} \vdash \overline{v} : S$ *and* $E; \overline{v} : S/\overline{p} \vdash \overline{v} \approx \overline{p}$

Proof. By induction on the structure of M and S.

1. If $M = \mathsf{Funsig}(v : M')M''$ then $M/p = M$ and the lemma trivially holds. For signatures ($M = \mathsf{Sig}\ \mathtt{\overline{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \mathtt{\overline{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$) let us first do the proofs of the properties (1a) … (1g). These proofs are done by induction on i.

   The base step is for $i = 1$ and it reduces to $E \vdash S_1 : \texttt{spec}$, $E \vdash \overline{p.\mathtt{v}_1} : S_1$ (true by **(Sig/Access)** from $E \vdash p : M$) and $E \vdash \overline{p.\mathtt{v}_1} : S_1/\overline{p.\mathtt{v}_1}$ (true by induction hypothesis (2)). Let us suppose that for $i = 1 \ldots k$ the properties hold and let us do the proof for $i = k + 1$.

   (a) Since $E \vdash \mathsf{Sig}\ \mathtt{\overline{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \mathtt{\overline{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End} : \texttt{modtype}$, by **(Sig/Access)** we obtain $E \vdash \overline{p.\mathtt{v}_j} : S_j\{\overline{v}_m \mapsto \overline{p.\mathtt{v}_m}\}$ for every $j = 1 \ldots n$ and by Lemma 4.1.1 $E; \overline{v}_1 : S_1 \ldots \overline{v}_{l-1} : S_{l-1} \vdash S_l : \texttt{spec}$ for every $l = 1 \ldots n$. Now, it is sufficient to use Lemma 4.1.5 l times and substitute $\overline{p.\mathtt{v}_1} \ldots \overline{p.\mathtt{v}_l}$ to $\overline{v}_1 \ldots \overline{v}_l$ respectively (from left to right) in order to get $E \vdash S_l\{\overline{v}_m \mapsto \overline{p.\mathtt{v}_m}\} : \texttt{spec}$ for every $l = 1 \ldots n$.

   (b) We have $E_{k-1/} \vdash S_k : \texttt{spec}$ by induction hypothesis (1c) and $E_{k-1/} \vdash \overline{p.\mathtt{v}_k} : S_k$ by induction hypothesis (1f). Hence by induction hypothesis (2) we get $E_{k-1/}; \overline{v}_k : S_k/\overline{p.\mathtt{v}_k} \vdash \overline{v}_k : S_k$ which is the desired conclusion.

   (c) Since $E \vdash \mathsf{Sig}\ \mathtt{\overline{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \mathtt{\overline{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End} : \texttt{modtype}$ we know that $E; \overline{v}_1 : S_1 \ldots \overline{v}_k : S_k \vdash S_{k+1} : \texttt{spec}$. By induction hypothesis (1b) we know that $E_{k/} \vdash \overline{v}_j : S_j$ for $j = 1 \ldots k - 1$ and we have just proved that $E_{k/} \vdash \overline{v}_k : S_k$. Hence we can use Lemma 4.1.7 and get $E_{k/} \vdash S_{k+1} : \texttt{spec}$.

   (d) By induction hypothesis (1c) we have $E_{k-1}/ \vdash S_k : \texttt{spec}$ and by induction hypothesis (1f) $E_{k-1/} \vdash \overline{p.\mathtt{v}_k} : S_k$. Now, we can use induction hypothesis of (2) and get $E_{k-1/}; \overline{v}_k : S_k/\overline{p.\mathtt{v}_k} \vdash \overline{v}_k \approx \overline{p.\mathtt{v}_k}$.

(e) In (1c) we have shown that $E_{k/} \vdash S_{k+1} : \texttt{spec}$. By (1a) and Lemma 4.1.4 we have also $E_{k/} \vdash S_{k+1}\{\overline{v}_j \mapsto \overline{p.\texttt{v}_j}\} : \texttt{spec}$. Since $E_{k/} \vdash \overline{v}_j \approx \overline{p.\texttt{v}_j}$ for every $j = 1 \ldots k$ (by (1d) and Lemma 4.1.4), we can apply Lemma 4.1.13 and get $E_{k/} \vdash S_{k+1} \approx S_{k+1}\{\overline{v}_j \mapsto \overline{p.\texttt{v}_j}\}$.

(f) By **(Sig/Access)** applied to $E \vdash \mathsf{Sig}\ \overline{\texttt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\texttt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End} : \texttt{modtype}$ we know that $E \vdash \overline{p.\texttt{v}_{k+1}} : S_{k+1}\{v \mapsto \overline{p.\texttt{v}}\}$. By Lemma 4.1.4 we have also $E_{k/} \vdash \overline{p.\texttt{v}_{k+1}} : S_{k+1}\{\overline{v}_j \mapsto \overline{p.\texttt{v}_j}\}$. In (1e) we have just shown that $E_{k/} \vdash S_{k+1} \approx S_{k+1}\{\overline{v}_j \mapsto \overline{p.\texttt{v}_j}\}$ and by symmetry of conversion (Lemma 4.1.12) $E_{k/} \vdash S_{k+1}\{\overline{v}_j \mapsto \overline{p.\texttt{v}_j}\} \approx S_{k+1}$. Consequently, we get $E_{k/} \vdash \overline{p.\texttt{v}_{k+1}} : S_{k+1}$ by Lemma 4.1.11.

(g) It is an easy corollary from (1f) and (2).

Now, let us show that for $M = \mathsf{Sig}\ \overline{\texttt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\texttt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$ the judgments $E \vdash M/p : \texttt{modtype}$ and $E \vdash M/p <: M$ hold. Note that we have $M/p = \mathsf{Sig}\ \overline{\texttt{v}}_1 \triangleright \overline{v}_1 : S_1/\overline{p.\texttt{v}_1} \ldots \overline{\texttt{v}}_n \triangleright \overline{v}_n : S_n/\overline{p.\texttt{v}_n}\ \mathsf{End}$.

By (1f) for every $i = 1 \ldots n$ we have $E_{i-1/} \vdash \overline{p.\texttt{v}_i} : S_i$. By induction hypothesis (2) this implies $E_{i-1/} \vdash S_i/\overline{p.\texttt{v}_i} : \texttt{spec}$ for every $i = 1 \ldots n$ and $E_{n/} \vdash \texttt{ok}$ by **(Env/Insert)**. By **(Sig/Form)** we obtain $E \vdash \mathsf{Sig}\ \overline{\texttt{v}}_1 \triangleright \overline{v}_1 : S_1/\overline{p.\texttt{v}_1} \ldots \overline{\texttt{v}}_n \triangleright \overline{v}_n : S_n/\overline{p.\texttt{v}_n}\ \mathsf{End} : \texttt{modtype}$.

To show that

$$E \vdash \mathsf{Sig}\ \overline{\texttt{v}}_1 \triangleright \overline{v}_1 : S_1/\overline{p.\texttt{v}_1} \ldots \overline{\texttt{v}}_n \triangleright \overline{v}_n : S_n/\overline{p.\texttt{v}_n}\ \mathsf{End} <: \mathsf{Sig}\ \overline{\texttt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\texttt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$$

it is sufficient that $E_{i/} \vdash \overline{v}_i : S_i$ for every $i = 1 \ldots n$, which is true by (1b).

2. There are two cases for specifications: abstract and manifest.

   (a) $S = \mathsf{Ty}(\Psi)$. By assumption we know that $E \vdash \mathsf{Ty}(\Psi) : \texttt{spec}$ (which implies $E \vdash \Psi : \Xi$) and $E \vdash \overline{p} : \mathsf{Ty}(\Psi)$ (which implies $E \vdash \overline{p} : \Psi$).

      i. For blocks and terms $S/\overline{p} = \mathsf{Eq}(\overline{p} : \Psi)$. Then $E \vdash \mathsf{Eq}(\overline{p} : \Psi) : \texttt{spec}$ holds by **(Eq/Form)**, $E \vdash \overline{p} : \mathsf{Eq}(\overline{p} : \Psi)$ by **(Eq/Sat)**, $E; \overline{v} : \mathsf{Eq}(\overline{p} : \Psi) \vdash \overline{v} : \mathsf{Ty}(\Psi)$ by **(Env/Lookup)** followed by **(Eq/Type)** and **(Ty/Sat)** and $E; \overline{v} : \mathsf{Eq}(\overline{p} : \Psi) \vdash \overline{v} \approx \overline{p}$ by **(Env/Lookup)** and **(Eq/Comp)**.

      ii. For modules $\Psi = M$ and $S/p = \mathsf{Eq}(p : M/p)$. Then $E \vdash \mathsf{Eq}(p : M/p) : \texttt{spec}$ holds because by induction hypothesis (1) we have $E \vdash M/p : \texttt{modtype}$ and $E \vdash p : \mathsf{Eq}(p : M/p)$ holds by strengthening of $E \vdash p : M$. The judgment $E; v : \mathsf{Eq}(p : M/p) \vdash v : \mathsf{Ty}(M)$ is true by **(Env/Lookup)** (for v) followed by **(Eq/Type)**, **(Subsumption)** (as $E \vdash M/p <: M$ by (1)) and **(Ty/Sat)**. As above, $E; v : \mathsf{Eq}(p : \Psi) \vdash v \approx p$ follows from **(Env/Lookup)** and **(Eq/Comp)**.

   (b) $S = \mathsf{Eq}(\phi : \Psi)$. By assumption we know that $E \vdash \mathsf{Eq}(\phi : \Psi) : \texttt{spec}$ (which implies $E \vdash \Psi : \Xi$ and $E \vdash \phi : \Psi$) and $E \vdash \overline{p} : \mathsf{Eq}(\phi : \Psi)$ (which implies $E \vdash \overline{p} : \Psi$ and $E \vdash \overline{p} \approx \phi$).

i. For blocks and terms $S/\overline{p} = \mathsf{Eq}(\phi : \Psi)$. The only nontrivial property to prove is $E; \overline{v} : \mathsf{Eq}(\phi : \Psi) \vdash \overline{v} \approx \overline{p}$. But it follows from symmetry and transitivity of $\approx$ for paths applied to the assumption $E \vdash \overline{p} \approx \phi$ and to $E; \overline{v} : \mathsf{Eq}(\phi : \Psi) \vdash \overline{v} \approx \phi$ (which follows from $E; \overline{v} : \mathsf{Eq}(\phi : \Psi) \vdash \overline{v} : \mathsf{Eq}(\phi : \Psi)$ by **(Eq/Comp)**).

ii. For modules $\phi = p'$, $\Psi = M$ and $S/p = \mathsf{Eq}(p' : M/p)$. To show that $E \vdash \mathsf{Eq}(p' : M/p) : \mathtt{spec}$ we need to know that $E \vdash M/p : \mathtt{modtype}$ and $E \vdash p' : M/p$. The former judgment follows from (1), since $E \vdash p : M$. The latter one follows by **(Subsumption)** from $E \vdash p' : M/p'$ (which is true by (1)) and $E \vdash M/p' <: M/p$ (which is true by Lemma 4.1.13 applied to $E \vdash M/p' : \mathtt{spec}$, $E \vdash M/p : \mathtt{spec}$ and $E \vdash p' \approx p$).
The judgment $E \vdash p : \mathsf{Eq}(p' : M/p)$ holds by **(Eq/Sat)**, because $E \vdash p : M/p$ from (1) and $E \vdash p \approx p'$.
The judgment $E; v : \mathsf{Eq}(p' : M/p) \vdash v : \mathsf{Eq}(p' : M)$ holds by **(Eq/Sat)**, because $E; v : \mathsf{Eq}(p' : M/p) \vdash v : M/p$ by **(Env/Lookup)** and **(Eq/Type)** and $E \vdash M/p <: M$ by induction hypothesis (1).
To prove that $E; v : \mathsf{Eq}(p' : M/p) \vdash v \approx p$ it is sufficient to use symmetry and transitivity of $\approx$ on $E \vdash p \approx p'$ (which is an assumption) and $E; v : \mathsf{Eq}(p' : M) \vdash v \approx p'$ (which follows from $E; v : \mathsf{Eq}(p' : M/p) \vdash v : \mathsf{Eq}(p' : M/p)$ by **(Eq/Comp)**.

□

The following lemma shows that our typing rules assign correct module types, block types and specifications to expressions. Note that we cannot prove the analogue fact for terms at this point. A simple inductive proof of this fact requires that all judgments are closed under arbitrary term substitutions, which is not true in our system. However, when we limit ourselves to principal derivations, the remaining correctness property of term typing can also be established (see Lemma 5.10.1).

Lemma 4.1.15. *For every environment E the following implications hold:*

- *if $E \vdash \phi : S$ then $E \vdash S : \mathtt{spec}$*
- *if $E \vdash P : BT$ then $E \vdash BT : \mathtt{blocktype}$*
- *if $E \vdash m : M$ then $E \vdash M : \mathtt{modtype}$*
- *if $E \vdash S_1 \approx S_2$ then $E \vdash S_1 : \mathtt{spec}$ and $E \vdash S_2 : \mathtt{spec}$*
- *if $E \vdash BT_1 \approx BT_2$ then $E \vdash BT_1 : \mathtt{blocktype}$ and $E \vdash BT_2 : \mathtt{blocktype}$*
- *if $E \vdash M_1 \approx M_2$ then $E \vdash M_1 : \mathtt{modtype}$ and $E \vdash M_2 : \mathtt{modtype}$*
- *if $E \vdash M_1 <: M_2$ then $E \vdash M_2 : \mathtt{modtype}$*

Proof. By induction on the derivation of the judgment considered. In most cases this follows directly from the premises of the typing rules and sometimes from the induction hypothesis. Let us see the rules where more work is needed:

For **(Env/Lookup)**

$$\frac{E_1; \overline{v} : S; E_2 \vdash \texttt{ok}}{E_1; \overline{v} : S; E_2 \vdash \overline{v} : S}$$

by Lemma 4.1.1 applied to $E_1; \overline{v} : S; E_2 \vdash \texttt{ok}$ we get $E_1 \vdash S : \texttt{spec}$ and by Lemma 4.1.4 $E_1; \overline{v} : S; E_2 \vdash S : \texttt{spec}$.

For **(Sig/Access)**

$$\frac{E \vdash p : \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}}{E \vdash \overline{p.\mathtt{v}_k} : S_k\{\overline{v}_j \mapsto \overline{p.\mathtt{v}_j}\}}$$

we have $E \vdash S_k\{\overline{v}_j \mapsto \overline{p.\mathtt{v}_j}\} : \texttt{spec}$ by Lemma 4.1.14 (1a).

For **(Funsig/Sub)**

$$\frac{E \vdash M_2' <: M_1' \qquad E; v : \mathsf{Ty}(M_2') \vdash M_1'' <: M_2''}{E \vdash \mathsf{Funsig}(v : M_1')M_1'' <: \mathsf{Funsig}(v : M_2')M_2''}$$

by induction hypothesis we have $E; v : \mathsf{Ty}(M_2') \vdash M_2'' : \texttt{modtype}$. By Lemma 4.1.2 in the derivation of $E; v : \mathsf{Ty}(M_2') \vdash M_2'' : \texttt{modtype}$ there is a subderivation of $E \vdash \mathsf{Ty}(M_2') : \texttt{spec}$. Since this judgment can be obtained only by **(Ty/Form)** rule, we have also $E \vdash M_2' : \texttt{modtype}$. From $E \vdash M_2' : \texttt{modtype}$ and $E; v : \mathsf{Ty}(M_2') \vdash M_2'' : \texttt{modtype}$ by **(Funsig/Form)** we get $E \vdash \mathsf{Funsig}(v : M_2')M_2'' : \texttt{modtype}$.

For **(Mod/App)**

$$\frac{E \vdash p : \mathsf{Funsig}(v : M_1)M_2 \qquad E \vdash p' : M_1}{E \vdash p\,p' : M_2\{v \mapsto p'\}}$$

by induction hypothesis $E \vdash \mathsf{Funsig}(v : M_1)M_2 : \texttt{modtype}$. Since the judgment of this form can be obtained only by **(Funsig/Form)** we have also $E; v : \mathsf{Ty}(M_1) \vdash M_2 : \texttt{modtype}$. By **(Ty/Sat)** $E \vdash p' : M_1$ implies $E \vdash p' : \mathsf{Ty}(M_1)$. Now, we can use Lemma 4.1.5 and get $E \vdash M_2\{v \mapsto p'\} : \texttt{modtype}$.

For **(Strengthening)**

$$\frac{E \vdash p : M}{E \vdash p : M/p}$$

we have $E \vdash M/p : \texttt{modtype}$ by Lemma 4.1.14 (1).

For **(Ty/Type)**

$$\frac{E \vdash P : \mathsf{Ty}(\Psi)}{E \vdash P : \Psi}$$

by induction hypothesis we have $E \vdash \mathsf{Ty}(\Psi) : \texttt{spec}$. Since this judgment can be obtained only by **(Ty/Form)**, we have also $E \vdash \Psi : \Xi$, which is $E \vdash \Psi : \texttt{modtype}$ for modules and $E \vdash \Psi : \texttt{blocktype}$ for blocks. □

4.2 Inversion

The following lemma shows that typing of module expressions can be inverted. This means in particular that structures, functors and module applications have principal types. Typing of the last kind of module expressions, access paths, is studied in detail in the next chapter.

Lemma 4.2.1. *Let $E \vdash m : M$.*

- *if* $m = \mathsf{Struct}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n := P_n\ \mathsf{End}$ *then*

  $$E \vdash m : SIG(m)$$

  and

  $$E \vdash SIG(m) <: M$$

  where $SIG(m) = \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$ *is the primary signature of* m*. Moreover,* $E; \overline{v}_1 : S_1 \ldots \overline{v}_{i-1} : S_{i-1} \vdash P_i : S_i$ *holds for all* i*.*

- *if* $m = \mathsf{Functor}[v : M_1]m'$ *then*

  $$E \vdash m : \mathsf{Funsig}(v : M_1)M_2$$

  for some M_2 *and*

  $$E \vdash \mathsf{Funsig}(v : M_1)M_2 <: M$$

  Moreover, $E; v : M_1 \vdash m' : M_2$ *holds.*

- *if* $m = p_1\, p_2$ *then*

  $$E \vdash p_1 : \mathsf{Funsig}(v : M_1)M_2$$

  for some M_1 *and* M_2 *such that* $E \vdash p_2 : M_1$ *and* $E \vdash M_2\{v \mapsto p_2\} <: M$*.*

Proof. It we examine the application of the rule which introduced the module expression m in the derivation of $E \vdash m : M$, it must have been **(Sig/Struct)**, **(Funsig/Functor)** or **(Mod/App)** depending on the syntax of m. After that rule only the **(Subsuption)** rule could have been applied. Using the premises of the syntax-oriented rule, the sequence of subtyping judgments used by subsumption rules and the fact that subtyping is reflexive and transitive (Lemmas 4.1.9 and 4.1.10), we get all the expected conclusions. □

Chapter 5

Principal typing

The typing system studied in this chapter is a sub-system of the one presented in Chapter 2. The derivations in this system are called *principal*, because they derive principal module types for module paths, as shown in Lemma 5.6.5.

Apart from typing rules for module paths, the system contains all typing rules for terms and elimination and computation rules for blocks. Lemma 5.8.7 shows that all valid term judgments in $\vdash$ are also derivable in $\vdash_P$, which means in particular that module subtyping is not essential for deriving term judgments.

The most important consequence of the latter property is the possibility to eliminate module entries from environments if the judgment considered does not concern modules (Lemmas 5.9.1 and 5.9.2). This operation is essential in the flattening phase of the module elimination process described in Chapter 3.

5.1 Paths

From all rules for environment and modules, we keep only those which serve to extract information from the environment or a signature. Note that the correctness of the environment in the first rule is established using the whole $\vdash$ system, and that there is no corresponding $E \vdash_P \texttt{ok}$ judgment.

$$\textbf{(P/Env/Lookup)} \qquad \frac{E_1; \overline{v} : S; E_2 \vdash \texttt{ok}}{E_1; \overline{v} : S; E_2 \vdash_P \overline{v} : S}$$

$$\textbf{(P/Sig/Access)} \qquad \frac{E \vdash_P p : \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}}{E \vdash_P \overline{p.\mathtt{v}_i} : S_i\{\overline{v}_i \mapsto \overline{p.\mathtt{v}_i}\}}$$

5.2 Specifications

The principal rules for specifications are also only those which say how to use specifications, i.e. elimination and computation rules.

Abstract specifications

$$\textbf{(P/Ty/Type)} \qquad \frac{E \vdash_P \overline{p} : \mathsf{Ty}(\Psi)}{E \vdash_P \overline{p} : \Psi}$$

Manifest specifications

$$\textbf{(P/Eq/Type)} \qquad \frac{E \vdash_P \overline{p} : \mathsf{Eq}(\phi : \Psi)}{E \vdash_P \overline{p} : \Psi}$$

$$\textbf{(P/Eq/Comp)} \qquad \frac{E \vdash_P \overline{p} : \mathsf{Eq}(\phi : \Psi)}{E \vdash_P \overline{p} \approx \phi}$$

where ϕ and Ψ are respectively: e and t, $\overline{p}$ and BT or p and M.

5.3 Terms

The rules in this section correspond to the rules for pure type systems. They are isomorphic to those given in Section 2.4.

$$\textbf{(P/Term/Prod)} \qquad \frac{E \vdash_P t_1 : \mathrm{s}_1 \quad E; v : \mathsf{Ty}(t_1) \vdash_P t_2 : \mathrm{s}_2}{E \vdash_P (v{:}t_1)\ t_2 : \mathrm{s}_3} \ \text{ if } (\mathrm{s}_1, \mathrm{s}_2, \mathrm{s}_3) \in \mathcal{R}$$

$$\textbf{(P/Term/Abs)} \qquad \frac{E; v : \mathsf{Ty}(t_1) \vdash_P e : t_2 \quad E \vdash_P (v{:}t_1)\ t_2 : \mathrm{s}}{E \vdash_P [v{:}t_1]e : (v{:}t_1)\ t_2}$$

$$\textbf{(P/Term/Axiom)} \qquad \frac{E \vdash \mathtt{ok}}{E \vdash_P \mathrm{s}_1 : \mathrm{s}_2} \ \text{ if } (\mathrm{s}_1, \mathrm{s}_2) \in \mathcal{A}$$

$$\textbf{(P/Term/App)} \qquad \frac{E \vdash_P e : (v{:}t_1)\ t_2 \quad E \vdash_P e' : t_1}{E \vdash_P e\ e' : t_2\{v \mapsto e'\}}$$

$$\textbf{(P/Term/Beta)} \qquad [v{:}t]e\ e' \longrightarrow_\beta e\{v \mapsto e'\}$$

$$\textbf{(P/Term/Conv)} \qquad \frac{E \vdash_P e : t \qquad E \vdash_P t' : \mathrm{s} \qquad E \vdash_P t \approx t'}{E \vdash_P e : t'}$$

5.4 Blocks

The principal typing rules for blocks contain again only elimination and computation rules.

Rewrite blocks In the forthcoming rules $E^\Gamma = v_1 : \mathsf{Ty}(t_1) \dots v_n : \mathsf{Ty}(t_n)$ and $R = \{E_i \mid e_i^L \longrightarrow e_i^R : \hat{t}_i\}_{i=1\dots m}$, where $E_i = v_1^i : \mathsf{Ty}(t_1^i); \dots ; v_{n_i}^i : \mathsf{Ty}(t_{n_i}^i)$

$$\textbf{(P/RewSpec/Fun)} \qquad \frac{E \vdash_P \overline{p} : \mathsf{RewSpec}(E^\Gamma, R)}{E \vdash_P p^j : t_j}$$

(P/RewSpec/Comp)

$$\frac{E \vdash_P \overline{p} : \mathsf{RewSpec}(E^\Gamma, R) \qquad \delta : E_i \to E}{E \vdash_P e_i^L \theta\delta \approx e_i^R \theta\delta} \quad \text{where} \quad \theta = \{v_j \mapsto p^j\}_{j=1\dots n}$$

The notation $\delta : E_i \to E$ means that for every $v : \mathsf{Ty}(t) \in E_i$ we have $E \vdash_P v\delta : t\delta$.

Inductive blocks Let $E^I = v_1^I : \mathsf{Ty}(t_1) \dots v_k^I : \mathsf{Ty}(t_k)$, $\quad E^C = v_1^C : \mathsf{Ty}(\hat{t}_1) \dots v_n^C : \mathsf{Ty}(\hat{t}_n)$

$$\textbf{(P/IndSpec/IndType)} \qquad \frac{E \vdash_P \overline{p} : \mathsf{IndSpec}(E^I := E^C)}{E \vdash_P p^j : t_j} \text{ for } j = 1 \dots k$$

$$\textbf{(P/IndSpec/Constr)} \qquad \frac{E \vdash_P \overline{p} : \mathsf{IndSpec}(E^I := E^C)}{E \vdash_P p^{k+i} : \hat{t}_i\{v_j^I \mapsto p^j\}_{j=1\dots k}} \text{ for } i = 1 \dots n$$

5.5 Convertibility

Convertibility rules in the principal system concern only terms and paths.

General

$$\textbf{(P/Conv/Trans)} \quad \frac{E \vdash_P \phi_1 \approx \phi_2 \quad E \vdash_P \phi_2 \approx \phi_3}{E \vdash_P \phi_1 \approx \phi_3} \qquad \textbf{(P/Conv/Sym)} \quad \frac{E \vdash_P \phi_1 \approx \phi_2}{E \vdash_P \phi_2 \approx \phi_1}$$

where ϕ_1, ϕ_2, ϕ_3 are terms or paths.

Paths

$$\textbf{(P/Conv/Refl)} \qquad \frac{E \vdash_P v : M}{E \vdash_P v \approx v}$$

$$\textbf{(P/Conv/Path)} \qquad \frac{E \vdash_P p.\mathtt{v} : \Psi \quad E \vdash_P p \approx p' \quad E \vdash_P p'.\mathtt{v} : \Psi'}{E \vdash_P p.\mathtt{v} \approx p'.\mathtt{v}}$$

where Ψ and Ψ' are module types or terms.

Terms

$$\textbf{(P/Conv/Beta)} \qquad \frac{E \vdash_P e_1 : t \quad e_1 =_\beta e_2 \quad E \vdash_P e_2 : t'}{E \vdash_P e_1 \approx e_2}$$

$$\textbf{(P/Conv/App)} \qquad \frac{E \vdash_P e_1\, e_2 : t \quad E \vdash_P e_1 \approx e_1' \quad E \vdash_P e_2 \approx e_2' \quad E \vdash_P e_1'\, e_2' : t'}{E \vdash_P e_1\, e_2 \approx e_1'\, e_2'}$$

(P/Conv/Prod)

$$\frac{E \vdash_P (v{:}t_1)\ t_2 : t \quad E \vdash_P t_1 \approx t_1' \quad E; v : \mathsf{Ty}(t_1) \vdash_P t_2 \approx t_2' \quad E \vdash_P (v{:}t_1')\ t_2' : t'}{E \vdash_P (v{:}t_1)\ t_2 \approx (v{:}t_1')\ t_2'}$$

(P/Conv/Abs)

$$\frac{E \vdash_P [v{:}t]e : \hat{t} \quad E \vdash_P t \approx t' \quad E; v : \mathsf{Ty}(t) \vdash_P e \approx e' \quad E \vdash_P [v{:}t']e' : \hat{t}'}{E \vdash_P [v{:}t]e \approx [v{:}t']e'}$$

5.6 Principal module types

The following facts help establish the correctness of principal typing. We show that a principal type of a module path is indeed principal: after strengthening it is a subtype of all other types the module path can have.

Lemma 5.6.1. *For every module type M, specification S and paths p_1 and p_2 such that $E \vdash p_1 \approx p_2$, the following two implications hold:*

- *if $E \vdash p_1 : M$ and $E \vdash p_2 : M$ then $E \vdash M/p_1 <: M/p_2$,*
- *if $E \vdash p_1 : S$ and $E \vdash p_2 : S$ then $E; v : S/p_1 \vdash v : S/p_2$.*

Proof. By induction on the structure of M and S respectively. If M is a functor type the result is trivial. Otherwise, M is a signature $\mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$. We have to show that $E \vdash M/p_1 : \mathtt{modtype}$, $E \vdash M/p_2 : \mathtt{modtype}$ which follow from Lemma 4.1.14 and

$$E; \overline{v}_1 : S_1/\overline{p_1.\mathtt{v}_1} \ldots \overline{v}_n : S_n/\overline{p_1.\mathtt{v}_n} \vdash \overline{v}_i : S_i/\overline{p_2.\mathtt{v}_i} \quad \text{for } i = 1 \ldots n$$

The latter can be easily obtained using Lemma 4.1.14, the rule **(Conv/Path)**, induction hypothesis and Lemma 4.1.4.

For specifications, the desired conclusion is immediate for term and block specifications and follows from induction hypothesis for module specifications. □

Lemma 5.6.2. *If $E \vdash p : M_1$ and $E \vdash M_1 <: M_2$ then $E \vdash M_1/p <: M_2/p$*

Proof. If M_1 is a functor type then so is M_2. In this case strengthening does nothing and the lemma is trivial. Otherwise, if M_1 is a signature $\mathsf{Sig}\ \overline{\mathtt{v}}_1 \rhd \overline{v}_1 : S_1 \dots \overline{\mathtt{v}}_n \rhd \overline{v}_n : S_n\ \mathsf{End}$, then M_2 is also a signature. Let $M_2 = \mathsf{Sig}\ \overline{\mathtt{v}'}_1 \rhd \overline{v'}_1 : S'_1 \dots \overline{\mathtt{v}'}_m \rhd \overline{v'}_m : S'_m\ \mathsf{End}$. By inversion of the rule **(Sig/Sub)** necessarily used to derive $E \vdash M_1 <: M_2$, we get $E \vdash M_1 : \mathtt{modtype}$, $E \vdash M_2 : \mathtt{modtype}$ and

$$E; \overline{v}_1 : S_1 \dots \overline{v}_n : S_n \vdash \overline{v}'_j : S'_j \quad \text{for } j = 1 \dots m$$

What we have to show is $E \vdash M_1/p : \mathtt{modtype}$, $E \vdash M_2/p : \mathtt{modtype}$ and

$$E; \overline{v}_1 : S_1/\overline{p.\mathtt{v}_1} \dots \overline{v}_n : S_n/\overline{p.\mathtt{v}_n} \vdash \overline{v}'_j : S'_j/\overline{p.\mathtt{v}'_j} \quad \text{for } j = 1 \dots m$$

The correctness of module types easily follows from Lemma 4.1.14 and **(Subsumption)**. To prove the correctness of the remaining judgments let us fix an arbitrary j. We present the proof for the case S'_j is a module specification.

Let us abbreviate $E; \overline{v}_1 : S_1 \dots \overline{v}_n : S_n$ by E_1 and $E; \overline{v}_1 : S_1/\overline{p.\mathtt{v}_1} \dots \overline{v}_n : S_n/\overline{p.\mathtt{v}_n}$ by $E_{1/}$. Similarly, let E_2 denote $E; \overline{v}'_1 : S'_1 \dots \overline{v}'_m : S'_m$ and $E_{2/}$ the environment $E; \overline{v}'_1 : S'_1/\overline{p.\mathtt{v}'_1} \dots \overline{v}'_m : S'_m/\overline{p.\mathtt{v}'_m}$. From correctness of corresponding signatures it follows that all four environments are correct.

If S'_j is an abstract specification $\mathsf{Ty}(M)$, then $S'_j/p.\mathtt{v}'_j = \mathsf{Eq}(p.\mathtt{v}'_j : M/p.\mathtt{v}'_j)$. Since $E_{1/}$ subsumes E_1, we know that $E_{1/} \vdash v'_j : \mathsf{Ty}(M)$. Using Lemma 4.1.14 for M_1 and M_2 followed by Lemma 4.1.4 and Lemma 4.1.7, we obtain $E_{1/} \vdash p.\mathtt{v}'_j : \mathsf{Ty}(M)$ and $E_{1/} \vdash v'_j \approx p.\mathtt{v}'_j$.

Now using $E_{1/} \vdash M/v'_j <: M/p.\mathtt{v}'_j$ by Lemma 5.6.1, we can show that $E_{1/} \vdash v'_j : M/p.\mathtt{v}'_j$. Using **(Eq/Sat)** we get $E_{1/} \vdash v'_j : \mathsf{Eq}(p.\mathtt{v}'_j : M/p.\mathtt{v}'_j)$.

If S'_j is a manifest specification $\mathsf{Eq}(p' : M)$ then $S'_j/p.\mathtt{v}'_j = \mathsf{Eq}(p' : M/p.\mathtt{v}'_j)$. We can show that $E_{1/} \vdash v'_j : M/p.\mathtt{v}'_j$ in the same way as before and the fact that $E_{1/} \vdash v'_j \approx p'$ follows from the assumption and Lemma 4.1.7. It is again enough to apply **(Eq/Sat)** to get the desired conclusion. $\square$

The lemma below shows that the principal type is correct for variables.

Lemma 5.6.3. *Let $S = \mathsf{Ty}(M)$ or $\mathsf{Eq}(p : M)$. If $E; v : S; E' \vdash v : M'$ then $E; v : S; E' \vdash M/v <: M'$.*

Proof. We prove by induction a slightly stronger fact: if $E; v : S; E' \vdash J$ where $J = v : M'$, $v : \mathsf{Ty}(M')$, $v : \mathsf{Eq}(p' : M')$ or $\mathsf{Eq}(v : M') : \mathtt{spec}$, then $E; v : S; E' \vdash M/v <: M'$.

We proceed by induction on the derivation of $E; v : S; E' \vdash J$. The rule **(Env/Lookup)** is covered by Lemma 4.1.14. For specification formation, introduction and elimination rules, we directly use induction hypothesis. The other rules which can assign a module type to a variable, **(Subsumption)** and **(Strengthening)**, are treated by Lemma 4.1.9 and Lemma 5.6.2 respectively. $\square$

Lemma 5.6.4. *Let $E \vdash p : M$ where $M = \mathsf{Sig}\ \overline{\mathtt{v}}_1 \rhd \overline{v}_1 : S_1 \dots \overline{\mathtt{v}}_n \rhd \overline{v}_n : S_n\ \mathsf{End}$ and $S_i = \mathsf{Ty}(M_i)$ or $\mathsf{Eq}(\hat{p} : M_i)$. If $E \vdash M <: \mathsf{Sig}\ \overline{\mathtt{v}'}_1 \rhd \overline{v'}_1 : S'_1 \dots \overline{\mathtt{v}'}_m \rhd \overline{v'}_m : S'_m\ \mathsf{End}$ and there exists j such that $\mathtt{v}'_j = \mathtt{v}_i$ then $S'_j = \mathsf{Ty}(M'_j)$ or $S'_j = \mathsf{Eq}(\hat{p}' : M'_j)$ where $E \vdash M_i\{\overline{v}_k \mapsto \overline{p.\mathtt{v}_k}\}/p.\mathtt{v}_i <: M'_j\{\overline{v}'_k \mapsto \overline{p.\mathtt{v}'_k}\}$.*

Proof. By inverting the subtyping judgment, we know that

$$E; \overline{v}_1 : S_1 \dots v_i : S_i \dots \overline{v}_n : S_n \vdash v_i : S'_j$$

By Lemma 5.6.3, we have

$$E; \overline{v}_1 : S_1 \dots v_i : S_i \dots \overline{v}_n : S_n \vdash M_i/v_i <: M'_j$$

and we conclude by applying n times Lemma 4.1.5, substituting $p.\mathtt{v}_k$ on v_k for all $k = 1 \dots n$. Of course $M_i\{\overline{v}_k \mapsto \overline{p.\mathtt{v}_k}\}/p.\mathtt{v}_i = M_i/v_i\{\overline{v}_k \mapsto \overline{p.\mathtt{v}_k}\}$. □

Lemma 5.6.5. *For every environment E, path p and module type M, if $E \vdash p : M$ then there exists exactly one module type M_P such that $E \vdash_P p : M_P$ and $E \vdash M_P/p <: M$.*

Proof. We use the same trick as in Lemma 5.6.3. We prove by induction on the derivation that if $E \vdash J$ where $J = p : M'$, $p : \mathsf{Ty}(M')$, $p : \mathsf{Eq}(p' : M')$ or $\mathsf{Eq}(p : M') : \mathtt{spec}$, then $E \vdash M_P/p <: M'$.

If the last rule was **(Env/Lookup)**, then we get the following $E; v : S; E' \vdash v : S$ where $S = \mathsf{Ty}(M)$ or $\mathsf{Eq}(p' : M)$. By **(P/Env/Lookup)** followed by **(P/Ty/Type)** or **(P/Eq/Type)**, we get $E; v : S; E' \vdash_P v : M$, and by Lemma 4.1.14 we know that $E; v : S; E' \vdash M/v <: M$.

Specification formation, satisfaction or elimination rules are immediately treated by induction hypothesis.

If the last rule was **(Sig/Access)**, we have:

$$\frac{E \vdash p : \mathsf{Sig}\ \overline{\mathtt{v}}'_1 \triangleright \overline{v}'_1 : S'_1 \dots \overline{\mathtt{v}}'_m \triangleright \overline{v}'_m : S'_m\ \mathsf{End}}{E \vdash p.\mathtt{v}'_j : S_j\{\overline{v}'_k \mapsto \overline{p.\mathtt{v}'_k}\}}$$

where $S'_j = \mathsf{Ty}(M'_j)$ or $\mathsf{Eq}(p' : M'_j)$. By induction hypothesis, we have $E \vdash_P p : M_P$ and $E \vdash M_P/p <: \mathsf{Sig}\ \overline{\mathtt{v}}'_1 \triangleright \overline{v}'_1 : S'_1 \dots \overline{\mathtt{v}}'_m \triangleright \overline{v}'_m : S'_m\ \mathsf{End}$. Consequently, M_P must be a signature $\mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \dots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$, having a module component $\mathtt{v}_i \triangleright v_i : S_i$ where $S_i = \mathsf{Ty}(M_i)$ or $\mathsf{Eq}(p'' : M_i)$ and $\mathtt{v_i} = \mathtt{v}'_\mathtt{j}$. We can use the rule **(P/Sig/Access)** followed by **(P/Ty/Type)** or **(P/Eq/Type)** to get $E \vdash_P p.\mathtt{v}_i : M_i\{\overline{v}_k \mapsto \overline{p.\mathtt{v}_k}\}$ and Lemma 5.6.4 to get $E \vdash_P M_i\{\overline{v}_k \mapsto \overline{p.\mathtt{v}_k}\}/p.\mathtt{v}_i <: M'_j\{\overline{v}_k \mapsto \overline{p.\mathtt{v}_k}\}$.

The rules **(Strengthenning)** and **(Subsumption)** are treated by Lemma 4.1.9 and Lemma 5.6.2 respectively. □

Lemma 5.6.6. *For every environment E, a path p and Ψ, which is a module type or term, if $E \vdash p : \Psi$ then $E \vdash_P p : \Psi'$ for some Ψ'.*

Proof. If $p = v$ then $E \vdash v : \Psi$ implies that there is a sequence of variables $\overline{v}'$ and a specification S such that $\overline{v}' : S \in E$ and $v \in \overline{v}'$. By **(P/Env/Lookup)** $E \vdash_P \overline{v}' : S$. Using **(P/Ty/Type)** or **(P/Eq/Type)** we get $E \vdash_P \overline{v} : \hat{\Psi}$. If $\hat{\Psi}$ is a block type, we apply **(P/RewSpec/Fun)**, **(P/IndSpec/IndType)** or **(P/IndSpec/Constr)** to get $E \vdash_P v : t$ for some term t. If $\hat{\Psi}$ is a module type or a term, we are done.

Otherwise $p = p'.\mathtt{v}_i$. In the derivation of $E \vdash p'.\mathtt{v}_i : \Psi$ there must be an instance of **(Sig/Access)** of the following form

$$\frac{E \vdash p' : \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}}{E \vdash \overline{p'.\mathtt{v}_i} : S'_i\{\overline{v}_j \mapsto \overline{p'.\mathtt{v}_j}\}}$$

By Lemma 5.6.5 we have $E \vdash_P p' : M_P$ and $E \vdash M_P/p' <: \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$. From the subtyping judgment it follows that M_P is a signature and that it has a label $\mathtt{v}_i$. Consequently, we can use **(P/Sig/Access)** and then **(P/Ty/Type)** or **(P/Eq/Type)** and optionally **(P/RewSpec/Fun)**, **(P/IndSpec/IndType)** or **(P/IndSpec/Constr)** to get $E \vdash_P p'.\mathtt{v}_i : \Psi'$ for some Ψ'. □

5.7 Properties of principal derivations

In this section we establish some basic metatheoretical properties of principal derivations, the most important of which is closure by term substitutions (Lemma 5.7.5). This lemma follows the pattern that is also followed by many other lemmas about operations transforming judgments $E \vdash_P J$ into $E' \vdash_P J'$: if $E \vdash J$ and $E' \vdash \mathtt{ok}$ then $E' \vdash J'$.

The proof is done by induction on the derivation of $E \vdash_P J$ by inspecting all possible typing rules. In order to use the induction hypothesis for premises of a given typing rule, one must assure that the corresponding environment is correct. Fortunately, in most cases this environment is simply equal E', and the other cases concern only rules for term abstraction and product, which are handled with the help of Lemmas 5.7.3 and 5.7.4.

Lemma 5.7.1. *For every environment E and judgment J, if $E \vdash_P J$ then $E \vdash J$.*

Proof. This is obvious, since the set of typing rules for $\vdash_P$ is a subset of the set of rules for $\vdash$. □

Lemma 5.7.2. *If $E_1; E_2 \vdash \mathtt{ok}$ and $E_1; E_3 \vdash_P J$ then $E_1; E_2; E_3 \vdash_P J$.*

Proof. By lemma 4.1.4, we know that $E_1; E_2; E_3 \vdash \mathtt{ok}$. By easy induction on the derivation of $E_1; E_3 \vdash_P J$, we get the desired conclusion. □

Lemma 5.7.3. *In every derivation of $E \vdash_P (v{:}t_1)\, t_2 : t$ there are subderivations of $E \vdash_P t_1 : \mathrm{s}_1$ and $E; v : \mathsf{Ty}(t_1) \vdash_P t_2 : \mathrm{s}_2$ for some sorts s_1 and s_2. Moreover, either $E \vdash t \approx \mathrm{s}_3$ or $t = \mathrm{s}_3$ for some s_3 such that $(\mathrm{s}_1, \mathrm{s}_2, \mathrm{s}_3) \in \mathcal{R}$.*

Proof. By induction on the derivation. The only typing rules that could result in the judgment $E \vdash_P (v{:}t_1)t_2 : t$ are product formation and conversion. In the first case, $E \vdash_P t_1 : \mathrm{s}_1$ and $E; v : \mathsf{Ty}(t_1) \vdash_P t_2 : \mathrm{s}_2$ are the premises and $t = \mathrm{s}_3$ for some sort s_3 such that $(\mathrm{s}_1, \mathrm{s}_2, \mathrm{s}_3) \in \mathcal{R}$. In the second case, we use the induction hypothesis. □

Lemma 5.7.4. *If $E \vdash_P [v{:}t]e : t'$ then $E \vdash_P t : \mathrm{s}$ for some sort s.*

Proof. Straightforward induction, just like in the previous lemma. □

Lemma 5.7.5. *Let E_1, E_2 and E_3 be environments such that $E_2 = v_1 : \mathsf{Ty}(t_1) \dots v_n : \mathsf{Ty}(t_n)$. Let δ be a substitution of the domain $\{v_1 \dots v_n\}$ such that $E_1 \vdash_P v_i\delta : t_i\delta$. If $E_1; E_3\delta \vdash \mathtt{ok}$ and $E_1; E_2; E_3 \vdash_P J$ then $E_1; E_3\delta \vdash_P J\delta$, unless J is of the form $v_i : S$ for some specification S and $i \in \{1 \dots n\}$.*

Proof. The proof is done by induction on the derivation of $E_1; E_2; E_3 \vdash_P J$.

If the last rule in the derivation is **(Env/Lookup)**

$$\frac{\hat{E}_1; \overline{v} : S; \hat{E}_2 \vdash \mathtt{ok}}{\hat{E}_1; \overline{v} : S; \hat{E}_2 \vdash_P \overline{v} : S}$$

where $\hat{E}_1; \overline{v} : S; \hat{E}_2 = E_1; E_2; E_3$, then there are two possibilities: either $\overline{v} : S \in E_2$ or $\overline{v} : S \notin E_2$. In the former case we have nothing to show since the conclusion has the form $v_i : S$ for some i. If $\overline{v} : S \notin E_2$ then we can use **(P/Env/Lookup)** on $E_1; E_3\delta \vdash \mathtt{ok}$ in order to get $E_1; E_3\delta \vdash_P \overline{v}\delta : S\delta$.

If the last rule in the derivation is **(P/Sig/Access)** then by induction hypothesis we know that $E_1; E_3\delta \vdash_P p\delta : (\mathsf{Sig}\ \overline{\mathrm{v}}'_1 \triangleright \overline{v}'_1 : S_1 \dots \overline{\mathrm{v}}'_{n'} \triangleright \overline{v}'_{n'} : S_{n'}\ \mathsf{End})\delta$. Since δ is a term substitution and p is a module path (starting with a module variable), $p\delta = p$. Consequently, we can use **(P/Sig/Access)** to obtain $E_1; E_3\delta \vdash \overline{p.\mathrm{v}'_k} : S_k\delta\{\overline{v}'_j \mapsto \overline{p.\mathrm{v}'_j}\}$ which is the desired conclusion as $S_k\delta\{\overline{v}'_j \mapsto \overline{p.\mathrm{v}'_j}\} = S_k\{\overline{v}'_j \mapsto \overline{p.\mathrm{v}'_j}\}\delta$.

For **(P/Ty/Type)**, by induction hypothesis we know that $E_1; E_3\delta \vdash_P \overline{p}\delta : \mathsf{Ty}(\Psi)\delta$. If $\overline{p}\delta = \overline{p}$ then we can use **(P/Ty/Type)** to get $E_1; E_3\delta \vdash_P \overline{p}\delta : \Psi\delta$. Otherwise $\overline{p} = v_i$ and $\Psi = t_i$. In that case $E_1; E_3\delta \vdash_P v_i\delta : t\delta$ by assuption and Lemma 5.7.2.

For **(P/Eq/Type)** and **(P/Eq/Comp)**, by induction hypothesis $E_1; E_3\delta \vdash_P \overline{p}\delta : \mathsf{Eq}(\phi : \Psi)\delta$ holds. Since variables that have abstract specification in the environment (like $v_i : \mathsf{Ty}(t_i) \in E_2$) cannot get manifest specifications in the principal system, we conclude that $\overline{p}\delta = \overline{p}$. Then by **(P/Eq/Type)** or **(P/Eq/Comp)** we get the desired conclusions.

For **(P/Term/Prod)**

$$\frac{E_1; E_2; E_3 \vdash_P t_1 : \mathrm{s}_1 \qquad E_1; E_2; E_3; v : \mathsf{Ty}(t_1) \vdash_P t_2 : \mathrm{s}_2}{E_1; E_2; E_3 \vdash_P (v : t_1)t_2 : \mathrm{s}_3}$$

by induction hypothesis we have $E_1; E_3\delta \vdash_P t_1\delta : \mathrm{s}_1$. To use induction hypothesis on $E_1; E_2; E_3; v : \mathsf{Ty}(t_1) \vdash_P t_2 : \mathrm{s}_2$ we need to know that $E_1; E_3\delta; v : \mathsf{Ty}(t_1\delta) \vdash \mathtt{ok}$. But $E_1; E_3\delta \vdash_P t_1\delta : \mathrm{s}_1$ implies $E_1; E_3\delta \vdash t_1\delta : \mathrm{s}_1$, and by **(Ty/Form)**, $E_1; E_3\delta \vdash \mathsf{Ty}(t_1\delta) : \mathtt{spec}$. Hence, by **(Env/Insert)** $E_1; E_3\delta; v : \mathsf{Ty}(t_1\delta) \vdash \mathtt{ok}$. Once we have $E_1; E_3\delta \vdash_P t_1\delta : \mathrm{s}_1$ and $E_1; E_3\delta; v : \mathsf{Ty}(t_1\delta) \vdash_P t_2\delta : \mathrm{s}_2$ we can use **(P/Term/Prod)** and get $E_1; E_3\delta \vdash_P (v : t_1\delta)t_2\delta : \mathrm{s}_3$.

For **(P/Term/Abs)** the situation is similar. By induction hypothesis we have $E_1; E_3\delta \vdash_P (v : t_1\delta)t_2\delta : \mathrm{s}_2$. By Lemma 5.7.3 $E_1; E_3\delta \vdash_P t_1\delta : \mathrm{s}$ for some sort s. Using this judgment, we prove $E_1; E_3\delta; v : \mathsf{Ty}(t_1\delta) \vdash \mathtt{ok}$ like above and consequently, by induction hypothesis, we get $E_1; E_3\delta; v : \mathsf{Ty}(t_1\delta) \vdash_P e\delta : t_2\delta$. Finally by **(P/Term/Abs)** we obtain $E_1; E_3\delta \vdash_P [v : t_1\delta]e\delta : (v : t_1\delta)t_2\delta$.

The proofs for other term rules are similar but simpler, since environments in the premises and in the conclusion are the same.

Let us now consider **(P/RewSpec/Fun)**

$$\frac{E_1; E_2; E_3 \vdash_P \overline{p} : \mathsf{RewSpec}(E^\Gamma, R)}{E_1; E_2; E_3 \vdash_P p^j : t_j}$$

We claim that $\overline{p}\delta = \overline{p}$. Indeed, let us suppose that $p^i\delta \neq p^i$ for some i. It means that $p^i = v.\mathtt{v}_1 \dots v_n$ and $v \in E_2$. Since E_2 is a term environment $p^i = v$. But in the principal system it is not possible that a variable satisfies a specification that is different from the one assigned to it in the environment (in particular, the only way to satisfy a $\mathsf{RewSpec}()$ is to deduce it from the environment). Hence, the assumption that $p^i\delta \neq p^i$ for some i is false and $\overline{p}\delta = \overline{p}$. Consequently, by **(P/RewSpec/Fun)** $E_1; E_3\delta \vdash_P p^j : t_j\delta$ which is the desired conclusion.

For **(P/RewSpec/Comp)**

$$\frac{E_1; E_2; E_3 \vdash_P \overline{p} : \mathsf{RewSpec}(E^\Gamma, R) \qquad \xi : \hat{E}_i \to E_1; E_2; E_3}{E_1; E_2; E_3 \vdash_P e_i^L\theta\xi \approx e_i^R\theta\xi} \text{ where } \theta = \{v'_j \mapsto p^j\}_{j=1\dots n}$$

where $E^\Gamma = v'_1 : \mathsf{Ty}(t_1) \dots v'_n : \mathsf{Ty}(t_n)$ and $R = \{\hat{E}_i \mid e_i^L \longrightarrow e_i^R : \hat{t}_i\}_{i=1\dots m}$, by induction hypothesis we know that $E_1; E_3\delta \vdash_P \overline{p}\delta : \mathsf{RewSpec}(E^\Gamma\delta, R\delta)$ and $\xi\delta : \hat{E}_i \to E_1; E_3\delta$. First, we deduce that $\overline{p}\delta = \overline{p}$, like above. Then, we can use **(P/RewSpec/Comp)** with $\xi\delta$ and $\theta\delta$ to get $E_1; E_3\delta \vdash_P (e_i^L\delta)(\theta\delta)(\xi\delta) \approx (e_i^R\delta)(\theta\delta)(\xi\delta)$ which is equivalent to $E_1; E_3\delta \vdash_P (e_i^L\theta\xi)\delta \approx (e_i^R\theta\xi)\delta$.

The proofs for **(P/IndSpec/Type)** and **(P/IndSpec/Constr)** are very similar to the one for **(P/RewSpec/Fun)**.

We are left with conversion rules. The proofs for **(P/Conv/Refl)**, **(P/Conv/Sym)**, **(P/Conv/Trans)**, **(P/Conv/Beta)**, **(P/Conv/Path)**, **(P/Conv/App)** are easy applications of induction hypotheses. In **(P/Conv/Refl)** and **(P/Conv/Path)** it is important that we have to do with module variables, which are not affected by the term substitution.

For **(P/Conv/Prod)**

$$\frac{\begin{array}{cc} E_1; E_2; E_3 \vdash_P (v : t_1)t_2 : t & E_1; E_2; E_3 \vdash_P (v : t'_1)t'_2 : t' \\ E_1; E_2; E_3 \vdash_P t_1 \approx t'_1 & E_1; E_2; E_3; v : \mathsf{Ty}(t_1) \vdash_P t_2 \approx t'_2 \end{array}}{E_1; E_2; E_3 \vdash_P (v : t_1)t_2 \approx (v : t'_1)t'_2}$$

by induction hypothesis we know that $E_1; E_3\delta \vdash_P (v : t_1\delta)t_2\delta : t\delta$, $E_1; E_3\delta \vdash_P (v : t'_1\delta)t'_2\delta : t'\delta$ and $E_1; E_3\delta \vdash_P t_1\delta \approx t'_1\delta$. By Lemma 5.7.3 $E_1; E_3\delta \vdash_P t_1\delta : \mathrm{s}$ for some sort s. Then also $E_1; E_3\delta \vdash t_1\delta : \mathrm{s}$ holds and by **(Ty/Form)** $E_1; E_3\delta \vdash \mathsf{Ty}(t_1\delta) : \mathtt{spec}$. By **(Env/Insert)** $E_1; E_3\delta; v : \mathsf{Ty}(t_1\delta) \vdash \mathtt{ok}$. Hence, by induction hypothesis $E_1; E_3\delta; v : \mathsf{Ty}(t_1\delta) \vdash_P t_2\delta \approx t'_2\delta$ and by **(P/Conv/Prod)** $E_1; E_3\delta \vdash_P (v : t_1\delta)t_2\delta \approx (v : t_1\delta)t'_2\delta$.

For **(P/Conv/Abs)** the proof is very similar to the one presented above, with one little exception that we use Lemma 5.7.4 and not 5.7.3 to show that the extended environment is correct. $\square$

Lemma 5.7.6. *If* $E_1; v : \mathsf{Ty}(t); E_2 \vdash_P J$, $E_1 \vdash_P t \approx t'$, $E_1 \vdash_P t : \mathrm{s}$ *and* $E_1 \vdash_P t' : \mathrm{s}'$ *for some sorts* s *and* s', *then* $E_1; v : \mathsf{Ty}(t'); E_2 \vdash_P J$.

Proof. By Lemma 5.7.2, $E_1; v' : \mathsf{Ty}(t'); v : \mathsf{Ty}(t); E_2 \vdash_P J$, where v' is a fresh variable. By Lemma 5.7.2 we have also $E_1; v' : \mathsf{Ty}(t') \vdash_P t \approx t'$ and using **(P/Env/Lookup)** and **(P/Ty/Type)** we have $E_1; v' : \mathsf{Ty}(t') \vdash_P v' : t'$. We can use conversion to obtain $E_1; v' : \mathsf{Ty}(t') \vdash_P v' : t$. Using Lemma 5.7.1 and specification satisfaction, we get $E_1; v' : \mathsf{Ty}(t') \vdash v' : \mathsf{Ty}(t)$. By Lemma 4.1.5 we get $E_1; v' : \mathsf{Ty}(t'); E_2\{v \mapsto v'\} \vdash \mathtt{ok}$. Now, by Lemma 5.7.5, $E_1; v' : \mathsf{Ty}(t'); E_2\{v \mapsto v'\} \vdash_P J\{v \mapsto v'\}$. Since v' was fresh, the latter judgment is α-convertible with the desired conclusion. ☐

5.8 Completeness of principal derivations

The goal of this section is to prove that all term judgments derivable in $\vdash$ are also derivable in $\vdash_P$. We will do the proof by induction on the derivation in $\vdash$ and therefore we need a very strong induction hypothesis about term information carried by non-term judgments. Therefore we define a function $\mathit{Essence}_P(\cdot)$, which assigns a set of term or path judgments in $\vdash_P$ to every judgment in $\vdash$. Informally $\mathit{Essence}_P(E \vdash J)$ is a complete information about paths and terms that can be deduced from $E \vdash J$ and expressed with principal judgments.

Before giving the definition of $\mathit{Essence}_P(\cdot)$ let us introduce an auxiliary measure function, called weight, which justifies the correctness of the definition of $\mathit{Essence}_P(\cdot)$.

Definition 5.8.1. *The* weight *of a term, block type, module type or specification is the number of all* Sig ... End*'s,* Ty*()'s and* Eq*()'s occurring in it.*

It is obvious that the weight is invariant under path substitution and that it is smaller (sometimes strictly) in a subexpression.

To see that the definition of $\mathit{Essence}_P(\cdot)$ is correct it is enough to remark, that for all judgments J of the form $\psi : \Phi$ the recursive call is done with Φ' of smaller weight.

Definition 5.8.2. *The* principal essence *of a judgment* $E \vdash J$, *written* $\mathit{Essence}_P(E \vdash J)$ *is the set of principal judgments of the form* $E; E' \vdash_P J'$ *defined as follows:*

$\mathit{Essence}_P(E \vdash e : t) =$
  $\{E \vdash_P e : t\}$

$\mathit{Essence}_P(E \vdash e_1 \approx e_2) =$
  $\{E \vdash_P e_1 \approx e_2\}$

$\mathit{Essence}_P(E \vdash p : \mathsf{Sig}\ \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}) =$
  $\bigcup\{\mathit{Essence}_P(E \vdash \overline{p.\mathsf{v}_i} : S_i\{\overline{v}_j \mapsto \overline{p.\mathsf{v}_j}\})\ \ \text{for } i = 1 \ldots n\}$

$\mathit{Essence}_P(E \vdash p_1 \approx p_2) =$
  $\{E \vdash_P p_1 \approx p_2\}$

$\mathit{Essence}_P(E \vdash \mathsf{Sig}\ \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End} <: \mathsf{Sig}\ \overline{\mathsf{v}'}_1 \triangleright \overline{v'}_1 : S'_1 \ldots \overline{\mathsf{v}'}_m \triangleright \overline{v'}_m : S'_m\ \mathsf{End}) =$
  $\bigcup\{\mathit{Essence}_P(E; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n \vdash \overline{v}'_i : S'_i)\ \ \text{for } i = 1 \ldots m\}$
  where $\{\overline{\mathsf{v}}_1 \ldots \overline{\mathsf{v}}_n\} \supseteq \{\overline{\mathsf{v}}'_1 \ldots \overline{\mathsf{v}}'_m\}$ *and* $\mathsf{v}_i^j = \mathsf{v'}_{i'}^{j'}$ *implies* $v_i^j = v'^{j'}_{i'}$

$$
\begin{array}{l}
Essence_P(E \vdash \mathsf{IndSpec}(E^I := E^C) : \mathtt{blocktype}) = \\
\quad \{E \vdash_P t_j : \mathrm{s}_j \ \ \textit{for } j = 1 \ldots k\} \cup \{E; E^I \vdash_P \hat{t}_i : \hat{\mathrm{s}}_i \ \ \textit{for } i = 1 \ldots n\} \\
\quad\ \textit{where } E^I = v_1^I : \mathsf{Ty}(t_1) \ldots v_k^I : \mathsf{Ty}(t_k) \quad E^C = v_1^C : \mathsf{Ty}(\hat{t}_1) \ldots v_n^C : \mathsf{Ty}(\hat{t}_n)
\end{array}
$$

$$
\begin{array}{l}
Essence_P(E \vdash \overline{p} : \mathsf{IndSpec}(E^I := E^C)) \\
\quad \{E \vdash_P p^j : t_j \ \ \textit{for } j = 1 \ldots k\} \cup \{E \vdash_P p^{k+i} : \hat{t}_i\{v_j^I \mapsto p^j\} \ \ \textit{for } i = 1 \ldots n\} \\
\quad\ \textit{where } E^I = v_1^I : \mathsf{Ty}(t_1) \ldots v_k^I : \mathsf{Ty}(t_k) \quad E^C = v_1^C : \mathsf{Ty}(\hat{t}_1) \ldots v_n^C : \mathsf{Ty}(\hat{t}_n)
\end{array}
$$

$$
\begin{array}{l}
Essence_P(E \vdash \mathsf{IndSpec}(E^I := E^C) \approx \mathsf{IndSpec}(E^{I'} := E^{C'})) = \\
\quad \{E \vdash_P t_j \approx t'_j \ \ \textit{for } j = 1 \ldots k\} \cup \{E; E^I \vdash_P \hat{t}_i \approx \hat{t'}_i \ \ \textit{for } i = 1 \ldots n\} \\
\quad\ \textit{where } \ E^I = v_1^I : \mathsf{Ty}(t_1) \ldots v_k^I : \mathsf{Ty}(t_k) \qquad E^{I'} = v_1^I : \mathsf{Ty}(t'_1) \ldots v_k^I : \mathsf{Ty}(t'_k) \\
\qquad\qquad\quad E^C = v_1^C : \mathsf{Ty}(\hat{t}_1) \ldots v_n^C : \mathsf{Ty}(\hat{t}_n) \quad E^{C'} = v_1^C : \mathsf{Ty}(\hat{t'}_1) \ldots v_n^C : \mathsf{Ty}(\hat{t'}_n)
\end{array}
$$

$$
\begin{array}{l}
Essence_P(E \vdash \mathsf{RewSpec}(E^\Gamma, R) : \mathtt{blocktype}) = \\
\quad \{E \vdash_P t_j : \mathrm{s}_j \ \ \textit{for } j = 1 \ldots n\} \cup \\
\quad \{E; v_1^i : \mathsf{Ty}(t_1^i) \ldots v_{j-1}^i : \mathsf{Ty}(t_{j-1}^i) \vdash_P t_j^i : \mathrm{s}_j^i \ \ \textit{for } j = 1 \ldots n_i\} \cup \\
\quad \{E; E^\Gamma; E_i \vdash_P \hat{t}_i : \hat{\mathrm{s}}_i \ \ \textit{for } i = 1 \ldots m\} \cup \\
\quad \{E; E^\Gamma; E_i \vdash_P e_i^L : \hat{t}_i \ \ \textit{for } i = 1 \ldots m\} \cup \\
\quad \{E; E^\Gamma; E_i \vdash_P e_i^R : \hat{t}_i \ \ \textit{for } i = 1 \ldots m\} \\
\quad\ \textit{where } \ E^\Gamma = v_1 : \mathsf{Ty}(t_1) \ldots v_n : \mathsf{Ty}(t_n) \\
\qquad\qquad\quad R = \{E_i \mid e_i^L \longrightarrow e_i^R : \hat{t}_i\}_{i=1 \ldots m} \\
\qquad\qquad\quad E_i = v_1^i : \mathsf{Ty}(t_1^i) \ldots v_{n_i}^i : \mathsf{Ty}(t_{n_i}^i)
\end{array}
$$

$$
\begin{array}{l}
Essence_P(E \vdash \overline{p} : \mathsf{RewSpec}(E^\Gamma, R)) = \\
\quad \{E \vdash_P p^j : t_j \ \ \textit{for } j = 1 \ldots n\} \cup \\
\quad \{E; E_i \vdash_P e_i^L\{v_j \mapsto p^j\} \approx e_i^R\{v_j \mapsto p^j\} \ \ \textit{for } i = 1 \ldots m\} \\
\quad\ \textit{where } E^\Gamma = v_1 : \mathsf{Ty}(t_1) \ldots v_n : \mathsf{Ty}(t_n) \quad R = \{E_i \mid e_i^L \longrightarrow e_i^R : \hat{t}_i\}_{i=1 \ldots m}
\end{array}
$$

$$
\begin{array}{l}
Essence_P(E \vdash \mathsf{RewSpec}(E^\Gamma, R) \approx \mathsf{RewSpec}(E^{\Gamma'}, R')) = \\
\quad \{E \vdash_P t_j \approx t'_j \ \ \textit{for } j = 1 \ldots n\} \cup \\
\quad \{E; v_1^i : \mathsf{Ty}(t_1^i) \ldots v_{j-1}^i : \mathsf{Ty}(t_{j-1}^i) \vdash_P t_j^i \approx t_j^{i\,\prime} \ \ \textit{for } i = 1 \ldots m \ \ \textit{for } j = 1 \ldots n_i\} \cup \\
\quad \{E; E^\Gamma; E_i \vdash_P e_i^L \approx e_i^{L'} \ \ \textit{for } i = 1 \ldots m\} \cup \\
\quad \{E; E^\Gamma; E_i \vdash_P e_i^R \approx e_i^{R'} \ \ \textit{for } i = 1 \ldots m\} \cup \\
\quad \{E; E^\Gamma; E_i \vdash_P \hat{t}_i \approx \hat{t}_i{}' \ \ \textit{for } i = 1 \ldots m\} \\
\quad\ \textit{where } \ E^\Gamma = v_1 : \mathsf{Ty}(t_1) \ldots v_n : \mathsf{Ty}(t_n) \quad E^{\Gamma'} = v_1 : \mathsf{Ty}(t'_1) \ldots v_n : \mathsf{Ty}(t'_n) \\
\qquad\qquad\quad R = \{E_i \mid e_i^L \longrightarrow e_i^R : \hat{t}_i\}_{i=1 \ldots m} \quad R' = \{E'_i \mid e_i^{L'} \longrightarrow e_i^{R'} : \hat{t}_i{}'\}_{i=1 \ldots m} \\
\qquad\qquad\quad E_i = v_1^i : \mathsf{Ty}(t_1^i) \ldots v_{n_i}^i : \mathsf{Ty}(t_{n_i}^i) \quad E'_i = v_1^i : \mathsf{Ty}(t_1^i{}') \ldots v_{n_i}^i : \mathsf{Ty}(t_{n_i}^i{}')
\end{array}
$$

$$
\begin{array}{l}
Essence_P(E \vdash \overline{p} : \mathsf{Ty}(\Psi)) = \\
\quad Essence_P(E \vdash \overline{p} : \Psi)
\end{array}
$$

$$
\begin{array}{l}
Essence_P(E \vdash \overline{p} : \mathsf{Eq}(\overline{\phi} : \Psi)) = \\
\quad \{E \vdash_P p^i \approx \phi^i \ \ \textit{for all } i\} \cup Essence_P(E \vdash \overline{p} : \Psi)
\end{array}
$$

$$
Essence_P(E \vdash J) = \emptyset \ \textit{otherwise}
$$

Lemma 5.8.1. *For any judgment $E \vdash J$, $Essence_P(E \vdash J)$ is a finite set of judgments of the form $E; E' \vdash_P J'$, where E' is a term environment, unless J is a subtyping judgment. Moreover the function $Essence_P(\cdot)$ is compatible with path substitutions, i.e. for a judgment $E; \overline{v} : S; E_1 \vdash J$ and a substitution $\delta = \{\overline{v} \mapsto \overline{p}\}$*

$$\begin{aligned} &Essence_P(E; E_1\delta \vdash J\delta) = \\ &\qquad \{E; E_1\delta; E'\delta \vdash_P J'\delta \quad | \quad E; \overline{v} : S; E_1; E' \vdash_P J' \in Essence_P(E; \overline{v} : S; E_1 \vdash J)\} \end{aligned}$$

Proof. By induction on the weight of Ψ in case $J = \phi : \Psi$ and by inspection of the definition of $Essence_P(\cdot)$. □

The final result of this section, Lemma 5.8.7, will be a proof by induction on the derivation of $E \vdash J$ that all judgments in $Essence_P(E \vdash J)$ are satisfied. However, some of the typing rules of $\vdash$ require a more involved proof. These are **(Env/Lookup)**, **(Subsumption)** and **(Strenghtenning)**. The following lemma is needed to handle the rule **(Env/Lookup)**.

Lemma 5.8.2. *For all judgments $E \vdash_P J$, if $E \vdash_P J$ holds then every judgment in $Essence_P(E \vdash J)$ is satisfied.*

Proof. Let us first consider judgments of the form $\phi : \Psi$. Since the definition of $Essence_P()$ follows elimination and computation rules, it is quite easy to see that the lemma is true for all such judgments, by simple induction on the weight of Ψ.

The only non-obvious judgment is $E \vdash_P \overline{p} : \mathsf{RewSpec}(E^\Gamma, R)$. First, using elimination, we obtain $E \vdash_P p_j : t_j$ for all $j = 1 \ldots n$. Second, for every $i = 1 \ldots m$, using Lemma 5.7.2, we derive $E; E_i \vdash_P \overline{p} : \mathsf{RewSpec}(E^\Gamma, R)$ and $E; E_i \vdash_P e_i^L\{v_j \mapsto p_j\} \approx e_i^R\{v_j \mapsto p_j\}$ using the identity substitution in the computation rule.

For judgments of the form $e_1 \approx e_2$ the lemma obviously holds. □

Lemma 5.8.3. *For every environment E, a path p and Ψ, which is a module type or term, if $E \vdash p : \Psi$ then $E \vdash_P p \approx p$.*

Proof. Let $p = v.\mathtt{v}_1 \ldots .\mathtt{v}_n$. We will show by induction on n that the lemma holds. Note that by Lemma 5.6.6 we have $E \vdash_P p : \Psi'$ for some Ψ'.

For $n = 0$, if v is a module variable then $E \vdash_P v \approx v$ follows from **(P/Conv/Refl)**. If v is a term variable then $E \vdash_P v \approx v$ follows from **(P/Conv/Beta)**.

Suppose now that $p = p'.\mathtt{v}_{n+1}$. From $E \vdash p : \Psi$ it follows by Lemma 4.1.3 that $E \vdash p' : \hat{\Psi}$ for some $\hat{\Psi}$. Hence, by induction hypothesis we get $E \vdash_P p' \approx p'$. Recall that we already have $E \vdash_P p'.\mathtt{v}_{n+1} : \Psi'$. Consequently, we can use **(P/Conv/Path)** to obtain $E \vdash p'.\mathtt{v}_{n+1} \approx p'.\mathtt{v}_{n+1}$, which is the desired conclusion. □

The next two lemmas are needed in the proof of Lemma 5.8.7 to handle the **(Subsumption)** rule for signatures. Lemma 5.8.5 states the required substitutivity property for $Essence_P(\cdot)$. However, since $Essence_P(\cdot)$ contains judgments in $\vdash_P$, we first prove by induction on the derivation of $E \vdash_P J$ that $Essence_P(E \vdash J)$ for such judgments is closed by substitution.

Lemma 5.8.4. *Let E and E_1 be environments. If $E; \overline{v} : S; E_1 \vdash_P J$ and $E \vdash \overline{p} : S$ and all judgments in $Essence_P(E \vdash \overline{p} : S)$ hold then all judgments in $Essence_P(E; E_1\delta \vdash J\delta)$ hold, where $\delta = \{\overline{v} \mapsto \overline{p}\}$.*

Proof. The proof is done by induction on the derivation of $E; \overline{v} : S; E_1 \vdash_P J$.

Let us first note that $E; \overline{v} : S; E_1 \vdash_P J$ implies $E; \overline{v} : S; E_1 \vdash J$ (by Lemma 5.7.1) and $E; \overline{v} : S; E_1 \vdash \mathtt{ok}$ (by Lemma 4.1.2). Consequently, by Lemma 4.1.5 we have $E; E_1\delta \vdash \mathtt{ok}$.

If the last rule in the derivation of $E; \overline{v} : S; E_1 \vdash_P J$ is **(P/Env/Lookup)**

$$\frac{E; \overline{v} : S; E_1 \vdash \mathtt{ok}}{E; \overline{v} : S; E_1; \vdash_P \overline{v}' : S'}$$

then there are two possibilities: either $\overline{v} = \overline{v}'$ or $\overline{v} \neq \overline{v}'$. If $\overline{v}'$ are different from $\overline{v}$ then we can simply use **(P/Env/Lookup)** on $E; E_1\delta \vdash \mathtt{ok}$ to get $E; E_1\delta \vdash_P \overline{v}'\delta : S'\delta$ if $\overline{v}' : S' \in E_1$ or $E; E_1\delta \vdash_P \overline{v}' : S'$ otherwise, and use Lemma 5.8.2 to get the desired conclusion.

If $\overline{v}' = \overline{v}$ then we can weaken the judgments in $Essence_P(E \vdash \overline{p} : S)$ by Lemma 5.7.2, to get the judgments in $Essence_P(E; E_1\delta \vdash \overline{p} : S)$, which is what we wanted since $\overline{v}\delta = \overline{p}$.

In case of the rule **(P/Sig/Access)**:

$$\frac{E; \overline{v} : S; E_1 \vdash_P p' : \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}}{E; \overline{v} : S; E_1; \vdash_P \overline{p'.\mathtt{v}_i} : S_i\{\overline{v}_j \mapsto \overline{p'.\mathtt{v}_j}\}}$$

we know that judgments in $Essence_P(E; E_1\delta \vdash p'\delta : (\mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End})\delta)$ are valid by induction hypothesis. By definition of $Essence_P(\cdot)$ the latter set is a superset of $Essence_P(E; E_1\delta \vdash \overline{p'\delta.\mathtt{v}_i} : S_i\delta\{\overline{v}_j \mapsto \overline{p'\delta.\mathtt{v}_j}\})$. Fortunately, since $p'\delta.\mathtt{v}_j = p'.\mathtt{v}_j\delta$, we can write the latter set as $Essence_P(E; E_1\delta \vdash \overline{p'.\mathtt{v}_i}\delta : S_i\{\overline{v}_j \mapsto \overline{p'.\mathtt{v}_j}\}\delta)$ which is the desired conclusion.

For **(P/Ty/Type)** the lemma trivially holds, since $Essence_P(E; E_1\delta \vdash \overline{p} : \Psi) = Essence_P(E; E_1\delta \vdash \overline{p} : \mathsf{Ty}(\Psi))$ which is true by induction hypothesis.

For **(P/Eq/Type)** and **(P/Eq/Comp)** the proof is similar since $Essence_P(E; E_1\delta \vdash \overline{p} : \Psi)$ and all $Essence_P(E; E_1\delta \vdash p^i \approx \phi^i)$ are included in $Essence_P(E; E_1\delta \vdash \overline{p} : \mathsf{Eq}(\phi : \Psi))$.

For almost all term rules, the lemma holds by simply using induction hypothesis. An exception is the rule **(P/Term/Axiom)** whose premise does not use principal typing $\vdash_P$. But we know already that $E; E_1\delta \vdash \mathtt{ok}$ and hence by **(P/Term/Axiom)** we get $E; E_1\delta \vdash_P \mathrm{s}_1 : \mathrm{s}_2$ which is the desired conclusion.

If the last rule in the derivation is **(P/RewSpec/Fun)** then the conclusion follows easily by induction hypothesis. Let us now consider the rule **(P/RewSpec/Comp)**:

$$\frac{E; \overline{v} : S; E_1 \vdash_P \overline{p}' : \mathsf{RewSpec}(E^\Gamma, R) \qquad \xi : \hat{E}_i \to E; \overline{v} : S; E_1}{E; \overline{v} : S; E_1 \vdash_P e_i^L\theta\xi \approx e_i^R\theta\xi} \text{ where } \theta = \{v_j' \mapsto p^{j'}\}_{j=1\ldots n}$$

where $E^\Gamma = v_1' : \mathsf{Ty}(t_1) \ldots v_n' : \mathsf{Ty}(t_n)$ and $R = \{\hat{E}_i \mid e_i^L \longrightarrow e_i^R : \hat{t}_i\}_{i=1\ldots m}$. By induction hypothesis, all judgments in $Essence_P(E; E_1\delta \vdash \overline{p}'\delta : \mathsf{RewSpec}(E^\Gamma\delta, R\delta))$ hold. In particular we have $E; E_1\delta; \hat{E}_i\delta \vdash_P e_i^L\{v_j' \mapsto p'^j\}\delta \approx e_i^R\{v_j' \mapsto p'^j\}\delta$.

From the induction hypothesis about $\xi : \hat{E}_i \to E; \overline{v} : S; E_1$, we get $\xi\delta : \hat{E}_i\delta \to E; E_1\delta$ i.e. $E; E_1\delta \vdash_P v^i_k\xi\delta : t^i_k\delta$ for all $k = 1...n_i$ (where $\hat{E}_i = v^i_1 : \mathsf{Ty}(t^i_1) \ldots v^i_{n_i} : \mathsf{Ty}(t^i_{n_i})$ is a term environment).

Thus, using Lemma 5.7.5, we get $E; E_1\delta \vdash_P e^L_i\{v'_j \mapsto p'^j\}\delta(\xi\delta) \approx e^R_i\{v'_j \mapsto p'^j\}\delta(\xi\delta)$. Since the domain of ξ and co-domain of δ are disjoint, the above is equivalent to

$$E; E_1\delta \vdash_P e^L_i\{v'_j \mapsto p'^j\}\xi\delta \approx e^R_i\{v'_j \mapsto p'^j\}\xi\delta$$

which is exactly what we have to prove.

For **(P/IndSpec/IndType)** and **(P/IndSpec/Constr)** the lemma easily holds by induction hypothesis.

Concerning conversion rules the only nontrivial proofs are for **(P/Conv/Refl)** and **(P/Conv/Path)**. For **(P/Conv/Refl)**

$$\frac{E; \overline{v} : S; E_1 \vdash_P v' : M}{E; \overline{v} : S; E_1 \vdash_P v' \approx v'}$$

v' is a module variable and either $v' = v$ or $v' \notin \overline{v}'$. In the latter case from $E; \overline{v} : S; E_1 \vdash v' : M$ we conclude $E; E_1\delta \vdash v' : M\delta$, by Lemma 4.1.5, and $E; E_1\delta \vdash_P v' : M'$ for some M', by Lemma 5.6.5. Hence, by **(P/Conv/Refl)** $E; E_1\delta \vdash v' \approx v'$. In the former case we have to show that $E; E_1\delta \vdash_P p \approx p$ where p satisfies $E \vdash p : S$ by assumption of the lemma. But from Lemma 5.8.3 we have already $E \vdash_P p \approx p$ and by Lemma 5.7.2 $E; E_1\delta \vdash_P p \approx p$.

For **(P/Conv/Path)**

$$\frac{E; \overline{v} : S; E_1 \vdash_P p'.\mathtt{v} : \Psi \quad E; \overline{v} : S; E_1 \vdash_P p' \approx p'' \quad E; \overline{v} : S; E_1 \vdash_P p''.\mathtt{v} : \Psi'}{E; \overline{v} : S; E_1 \vdash_P p.\mathtt{v} \approx p'.\mathtt{v}}$$

we want to show that $E; E_1\delta \vdash_P (p'.\mathtt{v})\delta \approx (p''.\mathtt{v})\delta$ follows from $\mathit{Essence}_P(E; E_1\delta \vdash (p'.\mathtt{v})\delta : \Psi\delta)$, $E; E_1\delta \vdash_P p'\delta \approx p''\delta$ and $\mathit{Essence}_P(E; E_1\delta \vdash (p''.\mathtt{v})\delta : \Phi'\delta)$. If $p'.\mathtt{v}$ and $p''.\mathtt{v}$ are terms then $\mathit{Essence}_P(E; E_1\delta \vdash (p'.\mathtt{v})\delta : \Psi\delta) = \{E; E_1\delta \vdash_P (p'.\mathtt{v})\delta : \Psi\delta\}$ and $\mathit{Essence}_P(E; E_1\delta \vdash (p''.\mathtt{v})\delta : \Psi'\delta) = \{E; E_1\delta \vdash_P (p''.\mathtt{v})\delta : \Psi'\delta\}$ and we can use **(P/Conv/Path)** to get $E; E_1\delta \vdash_P (p'.\mathtt{v})\delta \approx (p''.\mathtt{v})\delta$

For module paths, the premise $E; \overline{v} : S; E_1 \vdash_P p'.\mathtt{v} : \Psi$ implies $E; \overline{v} : S; E_1 \vdash p'.\mathtt{v} : \Psi$ and $E; E_1\delta \vdash (p'.\mathtt{v})\delta : \Psi\delta$ (by Lemma 4.1.5). Hence by Lemma 5.6.5 $E; E_1\delta \vdash_P (p'.\mathtt{v})\delta : \hat{\Psi}$ for some $\hat{\Psi}$. Similarly, we have $E; E_1\delta \vdash_P (p''.\mathtt{v})\delta : \hat{\Psi}$ for some $\hat{\Psi}'$ and we can use **(P/Conv/Path)** to get the desired conclusion. □

Lemma 5.8.5. *Let E and E_1 be environments and let $E; \overline{v} : S; E_1 \vdash J$ and $E \vdash \overline{p} : S$. If all judgments in $\mathit{Essence}_P(E; \overline{v} : S; E_1 \vdash J)$ and $\mathit{Essence}_P(E \vdash \overline{p} : S)$ hold then all judgments in $\mathit{Essence}_P(E; E_1\delta \vdash J\delta)$ hold, where $\delta = \{\overline{v} \mapsto \overline{p}\}$.*

Proof. By Lemma 5.8.1, we just have to show the validity of $E; E_1\delta; E'\delta \vdash_P J'\delta$ for all judgments $E; \overline{v} : S; E_1; E' \vdash_P J' \in \mathit{Essence}_P(E; \overline{v} : S; E_1 \vdash J)$.

By Lemma 5.8.4, all judgments in $\mathit{Essence}_P(E; E_1\delta; E'\delta \vdash J'\delta)$ hold and since J' is a term judgment, $\mathit{Essence}_P(E; E_1\delta; E'\delta \vdash J'\delta) = \{E; E_1\delta; E'\delta \vdash_P J'\delta\}$ and we are done. □

The following lemma is needed to handle the rule **(Strenghtenning)** in the proof of Lemma 5.8.7.

Lemma 5.8.6. *For every module type M and specification S the following implications hold:*

- *if $E \vdash p : M$ and all judgments in $Essence_P(E \vdash p : M)$ are valid then judgments in $Essence_P(E \vdash p : M/p)$ are also valid,*
- *if $E \vdash \overline{p} : S$ and all judgments in $Essence_P(E \vdash \overline{p} : S)$ are valid then judgments in $Essence_P(E \vdash \overline{p} : S/\overline{p})$ are also valid.*

Proof. By induction on the weight of M and S respectively. In case M is a functor type, $Essence_P(E \vdash p : M/p)$ is empty.

Otherwise M is a signature $\mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$, and $Essence_P(E \vdash p : M/p)$ is a sum of sets $Essence_P(E \vdash \overline{p.\mathtt{v}_i} : S_i/\overline{p.\mathtt{v}_i}\{\overline{v}_j \mapsto \overline{p.\mathtt{v}_j}\})$ for $i = 1 \ldots n$. Since $S_i/\overline{p.\mathtt{v}_i}\{\overline{v}_j \mapsto \overline{p.\mathtt{v}_j}\} = S_i\{\overline{v}_j \mapsto \overline{p.\mathtt{v}_j}\}/\overline{p.\mathtt{v}_i}$ judgments in all these sets are valid by induction hypothesis.

If S is an abstract specification $\mathsf{Ty}(\Psi)$ then $S/\overline{p}$ is $\mathsf{Eq}(\overline{p} : \Psi/\overline{p})$. In that case $Essence_P(E \vdash \overline{p} : S/\overline{p})$ is equal $\{E \vdash_P \overline{p} \approx \overline{p}\} \cup Essence_P(E \vdash \overline{p} : \Psi/\overline{p})$. The first part is true by Lemma 5.8.3 and the second is either equivalent to the assumption or follows from the induction hypothesis in case Ψ is a module type M.

If S is a manifest specification $\mathsf{Eq}(\phi : \Psi)$ the reasoning is very similar. □

Lemma 5.8.7. *If $E \vdash J$ then every judgment in $Essence_P(E \vdash J)$ is satisfied. In particular if $E \vdash e : t$ then $E \vdash_P e : t$.*

Proof. By induction on the derivation of $E \vdash J$.

If the last rule in the derivation of $E \vdash J$ is one of the following: **(Env/Empty)**, **(Env/Insert)**, **(Sig/Form)**, **(Sig/Struct)**, **(Funsig/Form)**, **(Funsig/Functor)**, **(Funsig/Sub)**, **(Mod/App)**, **(Ty/Form)**, **(Eq/Form)**, **(IndSpec/Ind)**, **(RewSpec/Rew)**, **(Conv/Ty)**, **(Conv/Eq)**, **(Conv/Sig)** or **(Conv/Funsig)**, then $Essence_P(E \vdash J) = \emptyset$ and the lemma trivially holds. Let us examine other rules.

Since **(Env/Lookup)** is also present in $\vdash_P$ (under the name **(P/Env/Lookup)**), we can use it to obtain $E \vdash_P \overline{v} : S$, and the satisfaction of all judgments from $Essence_P(E \vdash \overline{v} : S)$ follows from Lemma 5.8.2.

For **(Term/Prod)**, by induction hypothesis we have $Essence_P(E \vdash t_1 : \mathrm{s}_1) = \{E \vdash_P t_1 : \mathrm{s}_1\}$ and $Essence_P(E; v : \mathsf{Ty}(t_1) \vdash t_2 : \mathrm{s}_1) = \{E; v : \mathsf{Ty}(t_1) \vdash_P t_2 : \mathrm{s}_2\}$. Hence, we can use **(P/Term/Prod)** and get $E \vdash_P (v : t_1)t_2 : \mathrm{s}_3$ which is the only element of the desired conclusion $Essence_P(E \vdash (v : t_1)t_2 : \mathrm{s}_3)$. The proofs for other term rules are very similar to the one presented above.

For **(Sig/Access)** we want to know that all judgments in $Essence_P(E \vdash \overline{p.\mathtt{v}_k} : S_k\{\overline{v}_j \mapsto \overline{p.\mathtt{v}_j}\})$ hold. But this set is included in $Essence_P(E \vdash p : \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End})$ whose elements hold by induction hypothesis.

The case of **(Sig/Sub)** is even simpler, since the essence of the premise and of the conclusion of the rule are equal.

Let us now consider the rule **(Subsumption)**. If the module types appearing in this rule are functor types or a module expression is not a path, then the essence of the conclusion is empty. Otherwise the module expression is a path p and the module types are two signatures:

$$\frac{\begin{array}{c} E \vdash p : \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \dots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End} \\ E \vdash \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \dots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}\ <:\ \mathsf{Sig}\ \overline{\mathtt{v}'}_1 \triangleright \overline{v'}_1 : S'_1 \dots \overline{\mathtt{v}'}_m \triangleright \overline{v'}_m : S'_m\ \mathsf{End} \end{array}}{E \vdash p : \mathsf{Sig}\ \overline{\mathtt{v}'}_1 \triangleright \overline{v'}_1 : S'_1 \dots \overline{\mathtt{v}'}_m \triangleright \overline{v'}_m : S'_m\ \mathsf{End}}$$

From the induction hypothesis about the first premise we have the validity of all judgments from $Essence_P(E \vdash p : \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \dots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End})$ which is equal to the sum of the sets $Essence_P(E \vdash \overline{p.\mathtt{v}_i} : S_i\{\overline{v}_j \mapsto \overline{p.\mathtt{v}_j}\})$ for all $i = 1 \dots n$.

From the induction hypothesis about the subtyping judgment, we have the validity of $Essence_P(E; \overline{v}_1 : S_1 \dots \overline{v}_n : S_n \vdash \overline{v}'_i : S'_i)$ for all $i = 1 \dots m$. Since $E \vdash p.\mathtt{v}_1 : S_1$ and all judgments from $Essence_P(E \vdash p.\mathtt{v}_1 : S_1)$ are valid, using lemma 5.8.5, we can get any judgment in $Essence_P(E; \overline{v}_2 : S_2\{\overline{v}_1 \mapsto \overline{p.\mathtt{v}_1}\} \dots \overline{v}_n : S_n\{\overline{v}_1 \mapsto \overline{p.\mathtt{v}_1}\} \vdash \overline{v}'_i\{\overline{v}_1 \mapsto \overline{p.\mathtt{v}_1}\} : S'_i\{\overline{v}_1 \mapsto \overline{p.\mathtt{v}_1}\})$. We can continue substituting $\overline{p.\mathtt{v}_j}$ to $\overline{v}_j$ until $j = n$ and we obtain the validity of all judgments in $Essence_P(E \vdash \overline{p.\mathtt{v}'_i} : S'_i\{\overline{v}'_j \mapsto \overline{p.\mathtt{v}'_j}\})$ for any i, because $\overline{v}'_i\{\overline{v}_j \mapsto \overline{p.\mathtt{v}_j}\} = \overline{p.\mathtt{v}'_i}$ and $S'_i\{\overline{v}'_j \mapsto \overline{p.\mathtt{v}'_j}\} = S'_i\{\overline{v}_j \mapsto \overline{p.\mathtt{v}_j}\}$. Since the desired conclusion $Essence_P(E \vdash p : \mathsf{Sig}\ \overline{\mathtt{v}'}_1 \triangleright \overline{v'}_1 : S'_1 \dots \overline{\mathtt{v}'}_m \triangleright \overline{v'}_m : S'_m\ \mathsf{End})$ is equal to the sum of $Essence_P(E \vdash \overline{p.\mathtt{v}'_i} : S'_i\{\overline{v}'_j \mapsto \overline{p.\mathtt{v}'_j}\})$ for all $i = 1 \dots m$, we are done.

For the **(Strengthening)** rule, we simply apply Lemma 5.8.6.

For **(Ty/Sat)**, **(Ty/Type)**, **(Eq/Sat)**, **(Eq/Type)** and **(Eq/Comp)** it is easy to see that the essence of the conclusion is a subset of the essence of the premises. The same happens for **(IndSpec/Form)**, **(IndSpec/IndType)**, **(IndSpec/Constr)**, **(RewSpec/Form)**, **(RewSpec/Fun)** and **(RewSpec/Sat)**.

For **(RewSpec/Comp)** we want to show that $E \vdash_P e_i^L\theta\delta \approx e_i^R\theta\delta$ holds. By induction hypothesis we know that $E; E_i \vdash_P e_i^L\theta \approx e_i^R\theta$. But $\delta : E_i \to E$ is a term substitution and we can use Lemma 5.7.5 to get the desired conclusion.

For **(Block/Conv)**, there are two cases, for $\mathsf{IndSpec}$ and $\mathsf{RewSpec}$. We will examine the more difficult case for $\mathsf{RewSpec}$. From induction hypothesis, we have the validity of the following judgments:

$Essence_P(E \vdash \overline{p} : \mathsf{RewSpec}(E^\Gamma, R))$:

1. $\{E \vdash_P p^j : t_j \ \text{ for } j = 1 \dots n\}$

2. $\{E; E_i \vdash_P e_i^L\{v_j \mapsto p^j\} \approx e_i^R\{v_j \mapsto p^j\} \ \text{ for } i = 1 \dots m\}$

$Essence_P(E \vdash \mathsf{RewSpec}(E^\Gamma, R) \approx \mathsf{RewSpec}(E^{\Gamma'}, R'))$:

3. $\{E \vdash_P t_j \approx t'_j \ \text{ for } j = 1 \dots n\}$

4. $\{E; v^i_1 : \mathsf{Ty}(t^i_1) \dots v^i_{j-1} : \mathsf{Ty}(t^i_{j-1}) \vdash_P t^i_j \approx {t^i_j}' \;\text{ for } i = 1 \dots m \;\text{ for } j = 1 \dots n_i\}$

5. $\{E; E^\Gamma; E_i \vdash_P e^L_i \approx e^{L'}_i \;\text{ for } i = 1 \dots m\}$

6. $\{E; E^\Gamma; E_i \vdash_P e^R_i \approx e^{R'}_i \;\text{ for } i = 1 \dots m\}$

7. $\{E; E^\Gamma; E_i \vdash_P \hat{t}_i \approx \hat{t}_i{}' \;\text{ for } i = 1 \dots m\}$

   $\mathit{Essence}_P(E \vdash \mathsf{RewSpec}(E^{\Gamma'}, R') : \mathtt{blocktype})$:

8. $\{E \vdash_P t'_j : \mathrm{s}'_j \;\text{ for } j = 1 \dots n\}$

9. $\{E; v^i_1 : \mathsf{Ty}({t^i_1}') \dots v^i_{j-1} : \mathsf{Ty}({t^i_{j-1}}') \vdash_P {t^i_j}' : {\mathrm{s}^i_j}' \;\text{ for } j = 1 \dots n_i\}$

10. $\{E; E^{\Gamma'}; E_i{}' \vdash_P \hat{t}_i{}' : \hat{\mathrm{s}}_i{}' \;\text{ for } i = 1 \dots m\}$

11. $\{E; E^{\Gamma'}; E_i{}' \vdash_P e^{L'}_i : \hat{t}_i{}' \;\text{ for } i = 1 \dots m\}$

12. $\{E; E^{\Gamma'}; E_i{}' \vdash_P e^{R'}_i : \hat{t}_i{}' \;\text{ for } i = 1 \dots m\}$

where

$$\begin{array}{ll} E^\Gamma = v_1 : \mathsf{Ty}(t_1) \dots v_n : \mathsf{Ty}(t_n) & E^{\Gamma'} = v_1 : \mathsf{Ty}(t'_1) \dots v_n : \mathsf{Ty}(t'_n) \\ R = \{E_i \mid e^L_i \longrightarrow e^R_i : \hat{t}_i\}_{i=1\dots m} & R' = \{E'_i \mid e^{L'}_i \longrightarrow e^{R'}_i : \hat{t}_i{}'\}_{i=1\dots m} \\ E_i = v^i_1 : \mathsf{Ty}(t^i_1) \dots v^i_{n_i} : \mathsf{Ty}(t^i_{n_i}) & E'_i = v^i_1 : \mathsf{Ty}({t^i_1}') \dots v^i_{n_i} : \mathsf{Ty}({t^i_{n_i}}') \end{array}$$

Our goal is the validity of judgments in $\mathit{Essence}_P(E \vdash \overline{p} : \mathsf{RewSpec}(E^{\Gamma'}, R'))$, i.e.

$$\{E \vdash_P p^j : t'_j \;\text{ for } j = 1 \dots n\} \cup \{E; E'_i \vdash_P e^{L'}_i\{v_j \mapsto p^j\} \approx e^{R'}_i\{v_j \mapsto p^j\} \;\text{ for } i = 1 \dots m\}$$

The validity of the judgments from the first set can easily be obtained using judgments 1, 3 and 8 and the conversion rule.

Let us now consider the second set of judgments. First, from hypothesis 5 and 6 and Lemma 5.7.5, we obtain $E; E_i \vdash_P e^L_i\{v_j \mapsto p^j\} \approx e^{L'}_i\{v_j \mapsto p^j\}$ and $E; E_i \vdash_P e^R_i\{v_j \mapsto p^j\} \approx e^{R'}_i\{v_j \mapsto p^j\}$. Using transitivity, symmetry and hypothesis 2, we can show that $E; E_i \vdash_P e^{L'}_i\{v_j \mapsto p^j\} \approx e^{R'}_i\{v_j \mapsto p^j\}$. Now it is enough to use Lemma 5.7.6 n_i times to get the desired conclusion. Since the above reasoning can be carried on for any i, we are done.

We are left with conversion rules. The only ones for which the proof does not trivially hold by induction hypothesis are **(Conv/Refl)** and **(Conv/Path)**.

For **(Conv/Refl)**, the premise of the rule is $E \vdash v : M$. Since v is module variable, by Lemma 5.6.5 we have $E \vdash_P v : M'$ for some M'. Hence, by **(P/Conv/Refl)** $E \vdash_P v \approx v$ which is the desired conclusion.

For **(Conv/Path)**

$$\frac{E \vdash p.\mathtt{v} : \Psi \quad E \vdash p \approx p' \quad E \vdash p'.\mathtt{v} : \Psi'}{E \vdash p.\mathtt{v} \approx p'.\mathtt{v}}$$

we have two possibilities. If $p.\mathtt{v}$, $p'.\mathtt{v}$ are term paths then by induction hypothesis we have $E \vdash_P p.\mathtt{v} : \Psi$, $E \vdash_P p \approx p'$, $E \vdash_P p'.\mathtt{v} : \Psi'$ and by **(P/Conv/Path)** we get $E \vdash_P p.\mathtt{v} \approx p'.\mathtt{v}$, i.e. $Essence_P(E \vdash p.\mathtt{v} \approx p'.\mathtt{v})$. If $p.\mathtt{v}$, $p'.\mathtt{v}$ are module paths then by Lemma 5.6.5 we have $E \vdash_P p.\mathtt{v} : \hat{\Psi}$, $E \vdash_P p'.\mathtt{v} : \hat{\Psi}'$ for some $\hat{\Psi}$, $\hat{\Psi}'$ and again by **(P/Conv/Path)** we get the desired conclusion. □

5.9 Module elimination

The following two lemmas are the main goal of this chapter. They are used to prove Lemma 7.2.5 announced in Chapter 3 as Lemma 3.3.1. They both follow the pattern described in the beginning of Section 5.7.

Lemma 5.9.1. *Let $E = E_1; v : \mathsf{Ty}(M); E_2$, where E_2 does not contain module elements and $M = \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$. Let $E' = E_1; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n; E_2\delta$ where $\delta = \{\overline{v.\mathtt{v}_k} \mapsto \overline{v}_k\}$. If $E' \vdash \mathtt{ok}$ and $E \vdash_P J$ then $E' \vdash_P J\delta$ unless J is of the form $v : \mathsf{Ty}(M)$, $v : M$ or $v \approx v$.*

Proof. We do the proof by induction on the derivation of $E \vdash_P J$. Let us consider the rule **(P/Env/Lookup)**

$$\frac{\hat{E}_1; \overline{v}' : S'; \hat{E}_2 \vdash \mathtt{ok}}{\hat{E}_1; \overline{v}' : S'; \hat{E}_2 \vdash_P \overline{v}' : S'}$$

If $\overline{v}'$ is different from v, we simply use **(P/Env/Lookup)** to look up the specification of $\overline{v}'$ in E'. If $\overline{v}'$ comes from E_2 then S' is substituted in E' and if it comes from E_1 then $S' = S'\delta$, so in both situations we get $E' \vdash_P \overline{v}' : S'\delta$ as a conclusion. Of course $\overline{v}'\delta = \overline{v}'$.

For the rule **(P/Sig/Access)** the premise concerns either the variable v or some other path p. In the first case the premise is $E \vdash_P v : M$ and we have to show that

$$E' \vdash_P \overline{v.\mathtt{v}_i}\delta : S_i\{\overline{v}_j \mapsto \overline{v.\mathtt{v}_j}\}\delta$$

But the latter is equivalent to $E' \vdash_P \overline{v}_i : S_i$ which can be obtained using the rule **(P/Env/Lookup)**.

In case p is a different path, by induction hypothesis we get

$$E' \vdash_P p\delta : \mathsf{Sig}\ \overline{\mathtt{v}}'_1 \triangleright \overline{v}'_1 : S'_1\delta \ldots \overline{\mathtt{v}}'_m \triangleright \overline{v}'_m : S'_m\delta\ \mathsf{End}$$

Using **(P/Sig/Access)** we obtain

$$E' \vdash_P \overline{(p\delta).\mathtt{v}'_i} : S'_i\delta\{\overline{v}'_j \mapsto \overline{(p\delta).\mathtt{v}'_j}\}$$

which is equivalent to

$$E' \vdash_P \overline{p.\mathtt{v}'_i}\delta : S'_i\{\overline{v}'_j \mapsto p.\overline{\mathtt{v}}'_j\}\delta$$

because the codomain of δ, the set $\{\overline{v}_1 \ldots \overline{v}_n\}$, is disjoint with $\{\overline{v}'_1 \ldots \overline{v}'_m\}$.

The specification rules are easily handled by induction hypothesis.

The term rules are also easily handled with the exception of the rules **(P/Term/Prod)** and **(P/Term/Abs)**, where the environment in the premise is richer than the one in the conclusion. Let us start by **(P/Term/Abs)**:

$$\frac{E; v' : \mathsf{Ty}(t_1) \vdash_P e : t_2 \qquad E \vdash_P (v'{:}t_1)t_2 : \mathrm{s}}{E \vdash_P [v'{:}t_1]e : (v'{:}t_1)t_2}$$

Before using the first induction hypothesis, we must make sure $E'; v' : \mathsf{Ty}(t_1\delta) \vdash \mathtt{ok}$. However, the second induction hypothesis $E' \vdash_P (v'{:}t_1\delta)t_2\delta : \mathrm{s}$ lets us deduce $E' \vdash_P t_1\delta : \mathrm{s}_1$ by Lemma 5.7.3. This implies $E' \vdash t_1\delta : \mathrm{s}_1$ and hence $E' \vdash \mathsf{Ty}(t_1\delta) : \mathtt{spec}$ and $E'; v' : \mathsf{Ty}(t_1\delta) \vdash \mathtt{ok}$ by **(Ty/Form)** and **(Env/Insert)**. Now we can use both induction hypotheses and get the desired conclusion by **(P/Term/Abs)**.

For **(P/Term/Prod)** we do the same trick, but the sort of the domain type of the substituted product comes directly from the first induction hypothesis.

Block rules and conversion rules all follow from the induction hypothesis except for **(P/Conv/Prod)** and **(P/Conv/Abs)** where the richer substituted environment must be proved correct like for the rule **(P/Term/Abs)**, using Lemmas 5.7.3 and 5.7.4 respectively. □

Lemma 5.9.2. *Let $E = E_1; v : \mathsf{Ty}(M); E_2$, where E_2 does not contain module elements and $M = \mathsf{Funsig}(\hat{v} : M_1)M_2$. If $E_1; E_2 \vdash \mathtt{ok}$ and $E \vdash_P J$ then $E_1; E_2 \vdash_P J$ unless J is of the form $v : \mathsf{Ty}(M)$, $v : M$ or $v \approx v$.*

Proof. The proof is very similar to the previous one, but simpler. □

5.10 Correctness of types

The following lemma completes the correctness of types established by lemma 4.1.15.

Lemma 5.10.1. *Let E be an environment, e and t terms. If $E \vdash e : t$, then $E \vdash t : \mathrm{s}$ or $t = \mathrm{s}$ for some sort* s.

Proof. Since by lemma 5.8.7 all term judgments are also derivable in $\vdash_P$, it is sufficient to show that $E \vdash_P e : t$ implies $E \vdash t : \mathrm{s}$ or $t = \mathrm{s}$ for some sort s. Let us do the proof by induction on the derivation of $E \vdash_P e : t$

If the last rule in the derivation is **(P/Eq/Type)**

$$\frac{E \vdash_P p : \mathsf{Eq}(e : t)}{E \vdash_P p : t}$$

then by Lemma 5.7.1 we have $E \vdash p : \mathsf{Eq}(e : t)$ and by Lemma 4.1.15 $E \vdash \mathsf{Eq}(e : t) : \mathtt{spec}$. Since the judgment of that form can only be obtained by the **(Eq/Form)** rule, we know that $E \vdash t : \mathrm{s}$, for some sort s.

The proof for **(P/Ty/Type)** is very similar.

If the last rule in the derivation is **(P/Term/Prod)**, **(P/Term/Abs)**, **(P/Term/Axiom)** or **(P/Term/Conv)** the desired conclusion follows immediately from the premises of these rules. The situation is slightly more complicated for **(P/Term/App)**

$$\frac{E \vdash_P e : (v : t_1)t_2 \qquad E \vdash_P e' : t_1}{E \vdash_P e\, e' : t_2\{v \mapsto e'\}}$$

By induction hypothesis $E \vdash_P (v : t_1)t_2 : \mathrm{s}$ for some sort s and by Lemma 5.7.3 $E; v : \mathsf{Ty}(t_1) \vdash_P t_2 : \mathrm{s}'$ for some sort s'. Since $E \vdash_P e' : t_1$ we can use Lemma 5.7.5 and get $E \vdash_P t_2\{v \mapsto e'\} : \mathrm{s}'$.

If the last rule in the derivation is **(P/RewSpec/Fun)**

$$\frac{E \vdash_P \overline{p} : \mathsf{RewSpec}(E^\Gamma, R)}{E \vdash_P p^j : t_j}$$

then we know that $E \vdash \overline{p} : \mathsf{RewSpec}(E^\Gamma, R)$ and by Lemma 4.1.15 $E \vdash \mathsf{RewSpec}(E^\Gamma, R) :$ `blocktype`. Since the judgment of that form can be obtained only by **(RewSpec/Form)** rule, we know that $E \vdash t_j : \mathrm{s}_j$ for some sort s_j.

The proofs for **(P/IndSpec/IndType)** and **(P/IndSpec/Constr)** are very similar.

$\square$

Chapter 6

Module evaluation

In this chapter we describe a deterministic procedure evaluating all module applications in a structure. After defining all elements of the evaluation procedure (Section 6.1), we show that it preserves types in Section 6.2 and that it is terminating (Section 6.3). The final result of this chapter, Lemma 6.3.21, easily implies Lemma 3.2.1 from Chapter 3, completing the first part of the conservativity proof.

6.1 Definitions

The evaluation will take place from left to right in a textual representation of a structure. Since we need to establish properties of a structure evaluated up to a certain point, we need to define the notions of *prefix* and *suffix* of a structure as well as a set of *evaluation positions* denoting places where a structure can be cut into a prefix and a suffix.

6.1.1 Structure prefixes and suffixes

Definition 6.1.1. *Syntactic classes of* prefixes *and* suffixes *of a structure, denoted by Pre and Post respectively, are defined by the following grammar:*

$$\begin{array}{rcl} Pre & ::= & \epsilon \mid Pre\ \overline{\mathtt{v}}\triangleright\overline{v} : S := P \mid Pre\ \overline{\mathtt{v}}\triangleright\overline{v} : S := \underline{\mathsf{Struct}} \\ Post & ::= & \epsilon \mid \overline{\mathtt{v}}\triangleright\overline{v} : S := P\ Post \mid \underline{\mathsf{End}}\ Post \end{array}$$

To mark unbalanced Struct's in Pre and End's in $Post$, we use the underline $\underline{\mathsf{Struct}}$ and $\underline{\mathsf{End}}$. As an example, consider the following prefix and suffix:

$$\begin{array}{rcl} Pre & = & \mathtt{v}_1\triangleright v_1 : S_1 := P_1;\ \mathtt{v}_2\triangleright v_2 : S_2 := \underline{\mathsf{Struct}}\ \mathtt{v}_{21}\triangleright v_{21} : S_{21} := P_{21} \\ Post & = & \mathtt{v}_{22}\triangleright v_{22} : S_{22} := P_{22};\ \mathtt{v}_{23}\triangleright v_{23} : S_{23} := P_{23}\ \underline{\mathsf{End}};\ \mathtt{v}_3\triangleright v_3 : S_3 := P_3 \end{array}$$

Now, $\mathsf{Struct}\ Pre\ Post\ \mathsf{End}$ is the following syntactically correct structure:

Struct

$$
\begin{array}{l}
\quad \mathtt{v}_1 \triangleright v_1 : S_1 := P_1\\
\quad \mathtt{v}_2 \triangleright v_2 : S_2 := \mathsf{Struct}\\
\qquad\qquad\qquad \begin{array}{l} \mathtt{v}_{21} \triangleright v_{21} : S_{21} := P_{21}\\ \hline\hline \mathtt{v}_{22} \triangleright v_{22} : S_{22} := P_{22}\\ \mathtt{v}_{23} \triangleright v_{23} : S_{23} := P_{23} \end{array}\\
\qquad\qquad\quad \mathsf{End}\\
\quad \mathtt{v}_3 \triangleright v_3 : S_3 := P_3\\
\mathsf{End}
\end{array}
$$

The double horizontal bar denotes a place where the structure is divided into *Pre* and *Post*. In the text, the horizontal bar is replaced by the mark $||$ giving Struct *Pre* $||$ *Post* End. All possible places where a mark can be located in a structure are called evaluation positions:

Definition 6.1.2. *The set of* evaluation positions *in a structure*

$$m \;=\; \mathsf{Struct}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathtt{v}}_k \triangleright \overline{v}_k : S_k := P_k\ \mathsf{End}$$

denoted by $Ev(m)$, is defined as follows:

- *1, ..., $k+1$ are evaluation positions in m,*
- *if q is an evaluation position in m and*

  $$m|_q \;=\; \mathtt{v} \triangleright v : S := \mathsf{Struct}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n := P_n\ \mathsf{End}$$

  then $q \cdot 2 \cdot i$ for $i = 1 \ldots n+1$ are evaluation positions in m.

Note that for positions $q' = q \cdot 2 \cdot i$ where $i \leq n$ or $q' = i$ where $i \leq k$, $m|_{q'}$ is a valid component subexpression of m. In other cases, for $q' = q \cdot 2 \cdot n'$ with $n' = n + 1$ or for $q' = k + 1$, $m|_{q'}$ does not denote a valid component subexpression.

We extend the notation $BV(m, q')$ in the straightforward way to account also for those additional evaluation positions. Given a structure m as in the definition of evaluation positions,

- if $q' = k + 1$ then $BV(m, q') = \overline{v}_1 : S_1 \ldots \overline{v}_k : S_k$,
- if $q' = q \cdot 2 \cdot n'$ with $n' = n + 1$, then $BV(m, q') = BV(m, q); \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n$

By analogy to the definition of evaluated structure components and structures (Definition 3.1.1), we define evaluated structure prefixes.

Definition 6.1.3. *A structure prefix Pre is* evaluated *if all its complete structure components are evaluated.*

In other words, no path application can occur in *Pre* and for every structure component $\mathtt{v} \triangleright v : \mathsf{Ty}(M) := m$ occurring in *Pre*, m is evaluated and if it is a structure then M is its primary signature.

6.1.2 Value of a path

In order to evaluate a subexpression corresponding to a module application $p_1\, p_2$, we must determine the value of p_1. We search for this value in the context preceding the subexpression $p_1\, p_2$.

Definition 6.1.4. *Given an evaluated structure prefix Pre and a module path p, a* value *of the path p in Pre, denoted $val_p(Pre)$ is defined as follows:*

- *for a variable*

  $$val_v(Pre\ \mathtt{v}{\triangleright}v : \mathsf{Ty}(M) := m) \;=\; m$$

  $$val_v(Pre\ \mathtt{v}{\triangleright}v : \mathsf{Eq}(p : M) := m) \;=\; p$$

  $$val_v(Pre\ \overline{\mathtt{v}}'{\triangleright}\overline{v}' : S := P) \;=\; val_v(Pre) \quad \textit{if } v \neq \overline{v}'$$

  $$val_v(Pre\ \mathtt{v}'{\triangleright}v' : S := \underline{\mathsf{Struct}}) \;=\; val_v(Pre) \quad \textit{if } v \neq v'$$

- *for a complex path*

  $$val_{p.\mathtt{v}_i}(Pre) \;=\; m_i\{\overline{v}_j \mapsto \overline{p.\mathtt{v}_j}\}$$
  $$\textit{if } val_p(Pre) = \mathsf{Struct}\ \overline{\mathtt{v}}_1{\triangleright}\overline{v}_1 : S_1 := P_1 \ldots \mathtt{v}_i{\triangleright}v_i : \mathsf{Ty}(M) := m_i \ldots \overline{\mathtt{v}}_n{\triangleright}\overline{v}_n : S_n := P_n\ \mathsf{End}$$

  $$val_{p.\mathtt{v}_i}(Pre) \;=\; p'\{\overline{v}_j \mapsto \overline{p.\mathtt{v}_j}\}$$
  $$\textit{if } val_p(Pre) = \mathsf{Struct}\ \overline{\mathtt{v}}_1{\triangleright}\overline{v}_1 : S_1 := P_1 \ldots \mathtt{v}_i{\triangleright}v_i : \mathsf{Eq}(p' : M) := m_i \ldots \overline{\mathtt{v}}_n{\triangleright}\overline{v}_n : S_n := P_n\ \mathsf{End}$$

  $$val_{p.\mathtt{v}_i}(Pre) \;=\; p''.\mathtt{v}_i \quad \textit{if } val_p(Pre) = p''$$

The following lemma says that if a structure prefix is evaluated, the value of a path is typable by the principal type of the path.

Lemma 6.1.1. *Let $m = \mathsf{Struct}\ Pre \,||\, Post\ \mathsf{End}$ be a structure well-typed in the empty environment, where Pre is evaluated. Let $E = BV(m, q)$ where q is the position of the mark in m.*

If $E \vdash_P p : M$ then $E \vdash val_p(Pre) : M$ and $val_p(Pre)$ is either a path, a functor or a structure $\mathsf{Struct}\ \overline{\mathtt{v}}_1{\triangleright}\overline{v}_1 : S_1 := P_1 \ldots \overline{\mathtt{v}}_n{\triangleright}\overline{v}_n : S_n := P_n\ \mathsf{End}$. In the latter case the structure is evaluated and $M = \mathsf{Sig}\ \overline{\mathtt{v}}_1{\triangleright}\overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n{\triangleright}\overline{v}_n : S_n\ \mathsf{End}$.

Proof. By induction on the derivation of $E \vdash_P p : M$. The last rule is obviously **(P/Ty/Type)** or **(P/Eq/Type)**, and before that there is either **(P/Env/Lookup)** or **(P/Sig/Access)**. In the former case p is a variable and the conclusion is obvious.

In the latter case $p = p'.\mathtt{v}_i$ with $E \vdash_P p'.\mathtt{v}_i : M\{\overline{v}_j \mapsto \overline{p'.\mathtt{v}_j}\}$. The premise of the **(P/Sig/Access)** rule is $E \vdash_P p' : \mathsf{Sig}\ \overline{\mathtt{v}}_1{\triangleright}\overline{v}_1 : S_1 \ldots \mathtt{v}_i{\triangleright}v_i : S \ldots \overline{\mathtt{v}}_n{\triangleright}\overline{v}_n : S_n\ \mathsf{End}$. By induction hypothesis $m' = val_{p'}(Pre)$ has the following properties:

$$E \vdash m' : \mathsf{Sig}\ \overline{\mathtt{v}}_1{\triangleright}\overline{v}_1 : S_1 \ldots \mathtt{v}_i{\triangleright}v_i : \mathsf{Ty}(M) \ldots \overline{\mathtt{v}}_n{\triangleright}\overline{v}_n : S_n\ \mathsf{End}$$

where $S = \mathsf{Ty}(M)$ or $\mathsf{Eq}(\hat{p} : M)$ and m' is either a path p'', in which case $val_p(Pre) = p''.\mathtt{v}_i$ and $E \vdash p'' : M\{\overline{v}_j \mapsto \overline{p''.\mathtt{v}_j}\}$ or m' is an evaluated structure

$$\mathsf{Struct}\ \overline{\mathtt{v}}_1{\triangleright}\overline{v}_1 : S_1 := P_1 \ldots \mathtt{v}_i{\triangleright}v_i : S := m'' \ldots \overline{\mathtt{v}}_n{\triangleright}\overline{v}_n : S_n := P_n\ \mathsf{End}$$

In the latter case if $S = \mathsf{Ty}(M)$ then $val_p(Pre) = m''\{\overline{v}_j \mapsto \overline{p'.\mathtt{v}_j}\}$. By inversion

$$E; \overline{v}_1 : S_1 \ldots \overline{v}_{i-1} : S_{i-1} \vdash m'' : M$$

and given that $E \vdash \overline{p'.\mathtt{v}_k} : S_k\{\overline{v}_j \mapsto \overline{p'.\mathtt{v}_j}\}_{j=1\ldots k-1}$ for all $k = 1...n$, we can use $i-1$ times Lemma 4.1.5, to get

$$E \vdash m''\{\overline{v}_j \mapsto \overline{p'.\mathtt{v}_j}\} : M\{\overline{v}_j \mapsto \overline{p'.\mathtt{v}_j}\}$$

Since m' is evaluated, m'' is either a path, a functor or an evaluated structure. In the latter case M is its primary signature and hence $M\{\overline{v}_j \mapsto \overline{p'.\mathtt{v}_j}\}$ is the primary signature of $m''\{\overline{v}_j \mapsto \overline{p'.\mathtt{v}_j}\}$.

If $S = \mathsf{Eq}(\hat{p} : M)$ then $val_p(Pre) = \hat{p}\{\overline{v}_j \mapsto \overline{p'.\mathtt{v}_j}\}$. By correctness of the module type of p', we have $E; \overline{v}_1 : S_1 \ldots \overline{v}_{i-1} : S_{i-1} \vdash \hat{p} : M$ and hence $E \vdash \hat{p}\{\overline{v}_j \mapsto \overline{p'.\mathtt{v}_j}\} : M\{\overline{v}_j \mapsto \overline{p'.\mathtt{v}_j}\}$ as before. □

6.1.3 The evaluation function

The function $Eval(Pre \mid\mid Post)$ transforms a structure into an evaluated structure. The transformation is performed by small steps from left to right. The two parts — the evaluated Pre and not evaluated $Post$ — are separated by the mark $\mid\mid$ located on an evaluation position in $\mathsf{Struct}\ Pre\ Post\ \mathsf{End}$. For the sake of uniformity, we omit the outermost Struct and End in the definition of $Eval$.

There are four kinds of steps in the evaluation. A **simple** step just moves the mark over a term or block component or a module component which is already evaluated (but is not a structure). A **structure start** step moves the mark inside a sub-structure and **structure end** outside a sub-structure, correcting its exported module type. A **computation** step evaluates a module application using the function val defined in the previous section.

Definition 6.1.5. *The* evaluation function $Eval(Pre \mid\mid Post)$ *is defined by the following set of rewrite rules*

simple steps: $q \cdot i \longrightarrow q \cdot i'$, where $i' = i + 1$

$$Eval(Pre \mid\mid \mathtt{v}{\triangleright}v : \mathsf{Ty}(t) := e\ Post) \longrightarrow$$
$$Eval(Pre\ \mathtt{v}{\triangleright}v : \mathsf{Ty}(t) := e \mid\mid Post)$$

$$Eval(Pre \mid\mid \mathtt{v}{\triangleright}v : \mathsf{Eq}(e' : t) := e\ Post) \longrightarrow$$
$$Eval(Pre\ \mathtt{v}{\triangleright}v : \mathsf{Eq}(e' : t) := e \mid\mid Post)$$

$$Eval(Pre \mid\mid \overline{\mathtt{v}}{\triangleright}\overline{v} : \mathsf{Ty}(BT) := P\ Post) \longrightarrow$$
$$Eval(Pre\ \overline{\mathtt{v}}{\triangleright}\overline{v} : \mathsf{Ty}(BT) := P \mid\mid Post)$$

$$\begin{array}{l} \mathit{Eval}(\mathit{Pre} \mid\mid \mathsf{v}{\triangleright}v : \mathsf{Eq}(\overline{p} : BT) := P\ \mathit{Post}) \longrightarrow \\ \qquad \mathit{Eval}(\mathit{Pre}\ \mathsf{v}{\triangleright}v : \mathsf{Eq}(\overline{p} : BT) := P \mid\mid \mathit{Post}) \end{array}$$

$$\begin{array}{l} \mathit{Eval}(\mathit{Pre} \mid\mid \mathsf{v}{\triangleright}v : \mathsf{Ty}(M) := p\ \mathit{Post}) \longrightarrow \\ \qquad \mathit{Eval}(\mathit{Pre}\ \mathsf{v}{\triangleright}v : \mathsf{Ty}(M) := p \mid\mid \mathit{Post}) \end{array}$$

$$\begin{array}{l} \mathit{Eval}(\mathit{Pre} \mid\mid \mathsf{v}{\triangleright}v : \mathsf{Eq}(p : M) := m\ \mathit{Post}) \longrightarrow \\ \qquad \mathit{Eval}(\mathit{Pre}\ \mathsf{v}{\triangleright}v : \mathsf{Eq}(p : M) := m \mid\mid \mathit{Post}) \end{array}$$

$$\begin{array}{l} \mathit{Eval}(\mathit{Pre} \mid\mid \mathsf{v}{\triangleright}v : \mathsf{Ty}(M) := \mathsf{Functor}[\hat{v} : M']m\ \mathit{Post}) \longrightarrow \\ \qquad \mathit{Eval}(\mathit{Pre}\ \mathsf{v}{\triangleright}v : \mathsf{Ty}(M) := \mathsf{Functor}[\hat{v} : M']m \mid\mid \mathit{Post}) \end{array}$$

structure start: $q \cdot i \longrightarrow q \cdot i \cdot 2 \cdot 1$

$$\begin{array}{l} \mathit{Eval}(\mathit{Pre} \mid\mid \mathsf{v}{\triangleright}v : \mathsf{Ty}(M) := \mathsf{Struct}\ \overline{\mathsf{v}}_1{\triangleright}\overline{v}_1 : S_1 := P_1 \ldots \overline{\mathsf{v}}_n{\triangleright}\overline{v}_n : S_n := P_n\ \mathsf{End}\ \mathit{Post}) \longrightarrow \\ \qquad \mathit{Eval}(\mathit{Pre}\ \mathsf{v}{\triangleright}v : \mathsf{Ty}(M) := \underline{\mathsf{Struct}} \mid\mid \overline{\mathsf{v}}_1{\triangleright}\overline{v}_1 : S_1 := P_1 \ldots \overline{\mathsf{v}}_n{\triangleright}\overline{v}_n : S_n := P_n\ \underline{\mathsf{End}}\ \mathit{Post}) \end{array}$$

structure end: $q \cdot i \cdot 2 \cdot n' \longrightarrow q \cdot i'$ where $n' = n + 1$ and $i' = i + 1$

$$\begin{array}{l} \mathit{Eval}(\mathit{Pre}\ \mathsf{v}{\triangleright}v : \mathsf{Ty}(M) := \underline{\mathsf{Struct}}\ \overline{\mathsf{v}}_1{\triangleright}\overline{v}_1 : S_1 := P_1 \ldots \overline{\mathsf{v}}_n{\triangleright}\overline{v}_n : S_n := P_n \mid\mid \underline{\mathsf{End}}\ \mathit{Post}) \longrightarrow \\ \qquad \mathit{Eval}(\mathit{Pre}\ \mathsf{v}{\triangleright}v : \mathsf{Ty}(\mathsf{Sig}\ \overline{\mathsf{v}}_1{\triangleright}\overline{v}_1 : S_1 \ldots \overline{\mathsf{v}}_n{\triangleright}\overline{v}_n : S_n\ \mathsf{End}) := \\ \qquad\qquad\qquad\qquad \mathsf{Struct}\ \overline{\mathsf{v}}_1{\triangleright}\overline{v}_1 : S_1 := P_1 \ldots \overline{\mathsf{v}}_n{\triangleright}\overline{v}_n : S_n := P_n\ \mathsf{End} \mid\mid \mathit{Post}) \end{array}$$

computation: q does not change

$$\begin{array}{l} \mathit{Eval}(\mathit{Pre} \mid\mid \mathsf{v}{\triangleright}v : \mathsf{Ty}(M) := p_1\, p_2\ \mathit{Post}) \longrightarrow \\ \qquad \mathit{Eval}(\mathit{Pre} \mid\mid \mathsf{v}{\triangleright}v : \mathsf{Ty}(M) := m\{\hat{v} \mapsto p_2\}\ \mathit{Post}) \\ \qquad \text{if } \mathit{val}_{p_1}(\mathit{Pre}) = \mathsf{Functor}[\hat{v} : M']m \end{array}$$

$$\begin{array}{l} \mathit{Eval}(\mathit{Pre} \mid\mid \mathsf{v}{\triangleright}v : \mathsf{Ty}(M) := p_1\, p_2\ \mathit{Post}) \longrightarrow \\ \qquad \mathit{Eval}(\mathit{Pre} \mid\mid \mathsf{v}{\triangleright}v : \mathsf{Ty}(M) := p'\, p_2\ \mathit{Post}) \\ \qquad \text{if } \mathit{val}_{p_1}(\mathit{Pre}) = p' \end{array}$$

Lemma 6.1.2. *If* Pre_1 *is evaluated and* $\mathit{Eval}(\mathit{Pre}_1 \mid\mid \mathit{Post}_1) \longrightarrow \mathit{Eval}(\mathit{Pre}_2 \mid\mid \mathit{Post}_2)$ *then* Pre_2 *is evaluated.*

Proof. Obvious. □

6.2 Type preservation

In this section we show that every evaluation step preserves the module type of the structure. Given an evaluation step transforming a structure m_1 into m_2, we proceed in two phases. First, we show that specifications in m_2 are correct and subsume the correspondig specifications in m_1. Second, we prove that implementations in m_2 satisfy thier respective specifications.

Both lemmas are proved by inspecting all evaluation positions q' in m_1 or m_2. The reasonning depends on the relation between the position q' and the position of the mark in the evaluation step q (or $q \cdot i$). Different regions are depicted in Figure 6.1.

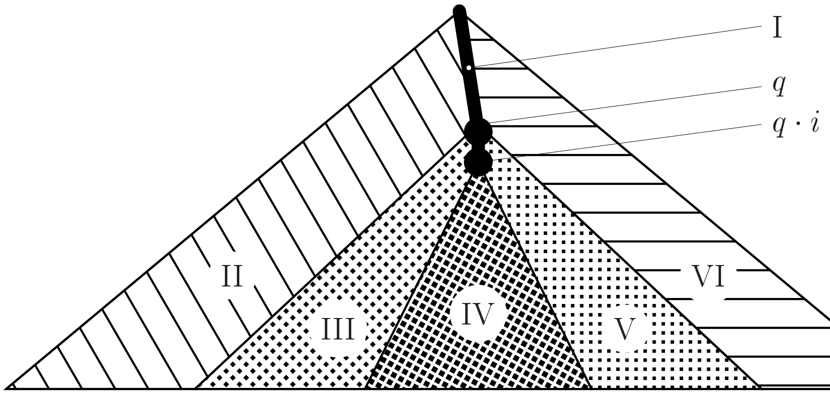

Figure 6.1: Regions relative to the positions q and $q \cdot i$

Lemma 6.2.1. *Let* $m_1 = \mathsf{Struct}\ Pre_1\ Post_1\ \mathsf{End}$ *and* $m_2 = \mathsf{Struct}\ Pre_2\ Post_2\ \mathsf{End}$ *and let* Pre_1 *be evaluated. If* $\vdash m_1 : \hat{M}$ *and* $Eval(Pre_1 \mid\mid Post_1) \longrightarrow Eval(Pre_2 \mid\mid Post_2)$ *then for every evaluation position* q' *in* m_1*, if* $BV(m_1, q') = \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n$ *then* q' *is an evaluation position in* m_2 *and* $BV(m_2, q')$ *subsumes* $BV(m_1, q')$ *i.e.* $BV(m_2, q') \vdash \overline{v}_i : S_i$ *for all* $i = 1 \ldots n$.

Proof. This is obvious for all **simple** steps and for **structure start**, as $m_1 = m_2$.

A **computation** step may increase the set of evaluation positions (in case functor application evaluates to a structure), but every evaluation position of m_1 is an evaluation position of m_2. Moreover since specifications do not change, corresponding bound environments are identical.

Let us now consider a **structure end** step, moving $\mid\mid$ from $q \cdot i \cdot 2 \cdot n'$ to $q \cdot i'$ for $i' = i + 1$. The easy cases cover q' of one of the following forms:

- q' is lexicographically smaller or equal $q \cdot i$ (regions I, II, III, q, $q \cdot i$ from Figure 6.1),
- q' is of the form $q \cdot i \cdot q''$ for some q'' (region IV),
- q' is lexicographically greater than q and q is not a prefix of q' (region VI).

In all of the above cases, the appropriate environments $BV(m_1, q')$ and $BV(m_2, q')$ are identical.

The remaining positions are those of the form $q \cdot j \cdot q''$ with $j > i$ (region V). In that case $BV(m_1, q') = E; v : \mathsf{Ty}(M); E'$ and

$$BV(m_2, q') = E; v : \mathsf{Ty}(\mathsf{Sig}\ \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}); E'$$

where $E = BV(m_1, q \cdot i) = BV(m_2, q \cdot i)$.

Since m_1 is well-typed and

$$m_1|_{q \cdot i} \;=\; \mathsf{v} \triangleright v : \mathsf{Ty}(M) := \mathsf{Struct}\ \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n := P_n\ \mathsf{End}$$

then

$$E \vdash \mathsf{Struct}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n := P_n\ \mathsf{End} : M$$

and hence by inversion

$$E \vdash \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End} : \mathtt{modtype}$$

and

$$E \vdash \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End} <: M$$

Consequently

$$E; v : \mathsf{Ty}(\mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}) \vdash \mathtt{ok}$$

and also

$$E; v : \mathsf{Ty}(\mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}) \vdash v : \mathsf{Ty}(M)$$

Using Lemma 4.1.7, we obtain $E; v : \mathsf{Ty}(\mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}); E' \vdash \mathtt{ok}$ and the subsumption obviously holds. $\square$

Lemma 6.2.2. *Let* $m_1 = \mathsf{Struct}\ \mathit{Pre}_1\ \mathit{Post}_1\ \mathsf{End}$ *and* $m_2 = \mathsf{Struct}\ \mathit{Pre}_2\ \mathit{Post}_2\ \mathsf{End}$*. If* $\vdash m_1 : \hat{M}$ *and and* $\mathit{Eval}(\mathit{Pre}_1 \mid\mid \mathit{Post}_1) \longrightarrow \mathit{Eval}(\mathit{Pre}_2 \mid\mid \mathit{Post}_2)$ *then for every evaluation position* q' *in* m_2*, such that* $m_2|_{q'} = \overline{\mathtt{v}} \triangleright \overline{v} : S := P$*, we have* $BV(m_2, q') \vdash P : S$*.*

Proof. This is obvious for all simple steps and for a **structure start** step, as $m_1 = m_2$.

For a **computation** step performed at position q, if q' verifies one of the following conditions:

- q' is lexicographically smaller that q but is not a prefix of q (region II in Figure 6.1),
- q' is lexicographically greater than q but q is not a prefix of q' (region VI),

the subterms $m_2|_{q'}$ and $m_1|_{q'}$ are identical and moreover $BV(m_1, q') = BV(m_2, q')$ so $BV(m_2, q') \vdash P : S$ follows immediately from the corresponding fact resulting from the correctness of m_1.

Now let us consider evaluation positions q' which have prefix q (region III-V). First, let $q' = q$. Let us remind, that $m_1|_q = \mathtt{v} \triangleright v : \mathsf{Ty}(M) := p_1\, p_2$ and $m_2|_q$ equals either $\mathtt{v} \triangleright v : \mathsf{Ty}(M) := p'\, p_2$ if $\mathit{val}_{p_1}(\mathit{Pre}) = p'$ or $\mathtt{v} \triangleright v : \mathsf{Ty}(M) := m\{\hat{v} \mapsto p_2\}$ if $\mathit{val}_{p_1}(\mathit{Pre}) = \mathsf{Functor}[\hat{v} : M']m$.

Let us denote the environment $BV(m_1, q) = BV(m_2, q)$ by E. From the correctness of m_1, it follows that $E \vdash p_1\, p_2 : M$. Using inversion and principal typing properties, we obtain $E \vdash_P p_1 : \mathsf{Funsig}(\hat{v} : M_1)M_2$ and $E \vdash p_2 : M_1$ and $E \vdash M_2\{\hat{v} \mapsto p_2\} <: M$. By Lemma 6.1.1, $E \vdash \mathit{val}_{p_1}(\mathit{Pre}) : \mathsf{Funsig}(\hat{v} : M_1)M_2$.

If $\mathit{val}_{p_1}(\mathit{Pre}) = p'$, then obviously $E \vdash p'\, p_2 : M_2\{\hat{v} \mapsto p_2\}$ and $E \vdash p'\, p_2 : M$ by subsumption and $E \vdash p'\, p_2 : \mathsf{Ty}(M)$ by specification satisfaction.

If $\mathit{val}_{p_1}(\mathit{Pre}) = \mathsf{Functor}[\hat{v} : M']m$ then by inversion $E \vdash M_1 <: M'$ and $E; \hat{v} : \mathsf{Ty}(M') \vdash m : M_2$. Since $E \vdash p_2 : M_1$, we have $E \vdash p_2 : M'$ and by Lemma 4.1.5, $E \vdash m\{\hat{v} \mapsto p_2\} : M_2\{\hat{v} \mapsto p_2\}$. By subsumption and specification satisfaction, $E \vdash m\{\hat{v} \mapsto p_2\} : \mathsf{Ty}(M)$.

If $q \neq q'$ and $p_1\, p_2$ evaluates to a structure, q' may be of the form $q \cdot 2 \cdot q''$ for some q''. The correctness of the corresponding $m_2|_{q'} = m\{\hat{v} \mapsto p_2\}|_{q''}$ follows from the correctness of $m\{\hat{v} \mapsto p_2\}$.

To end up the computation step we are left with the positions q' which are prefixes of q (region I). We delay the consideration of this case until we study the **structure end** step.

Consider now a **structure end** step moving the mark $||$ from $q \cdot i \cdot 2 \cdot n'$ to $q \cdot i'$ where $i' = i + 1$. If q' satisfies one of the following conditions:

- q' is lexicographically smaller than $q \cdot i$ but is not a prefix of $q \cdot i$ (regions II, III)
- q' is of the form $q \cdot i \cdot q''$ for some $q'' \neq \epsilon$ (region IV),
- q' is lexicographically greater than q but q is not a prefix of q' (region VI),

the corresponding subterms and bound environments are identical.

Now let us consider $q' = q \cdot i$. Let us denote $BV(m_1, q') = BV(m_2, q')$ by E. Since m_1 is well-typed and

$$m_1|_{q \cdot i} \;=\; \mathsf{v}{\triangleright}v : \mathsf{Ty}(M) := \mathsf{Struct}\ \overline{\mathsf{v}}_1{\triangleright}\overline{v}_1 : S_1 := P_1 \ldots \overline{\mathsf{v}}_n{\triangleright}\overline{v}_n : S_n := P_n\ \mathsf{End}$$

then

$$E \vdash \mathsf{Struct}\ \overline{\mathsf{v}}_1{\triangleright}\overline{v}_1 : S_1 := P_1 \ldots \overline{\mathsf{v}}_n{\triangleright}\overline{v}_n : S_n := P_n\ \mathsf{End} : M$$

and hence by inversion

$$E \;\vdash\; \mathsf{Struct}\ \overline{\mathsf{v}}_1{\triangleright}\overline{v}_1 : S_1 := P_1 \ldots \overline{\mathsf{v}}_n{\triangleright}\overline{v}_n : S_n := P_n\ \mathsf{End} \;:\; \mathsf{Sig}\ \overline{\mathsf{v}}_1{\triangleright}\overline{v}_1 : S_1 \ldots \overline{\mathsf{v}}_n{\triangleright}\overline{v}_n : S_n\ \mathsf{End}$$

So since

$$\begin{aligned} m_2|_{q \cdot i} = \mathsf{v}{\triangleright}v &: \mathsf{Ty}(\mathsf{Sig}\ \overline{\mathsf{v}}_1{\triangleright}\overline{v}_1 : S_1 \ldots \overline{\mathsf{v}}_n{\triangleright}\overline{v}_n : S_n\ \mathsf{End}) \\ &:= \mathsf{Struct}\ \overline{\mathsf{v}}_1{\triangleright}\overline{v}_1 : S_1 := P_1 \ldots \overline{\mathsf{v}}_n{\triangleright}\overline{v}_n : S_n := P_n\ \mathsf{End} \end{aligned}$$

this structure component is correct.

For all evaluation positions q' of the form $q' = q \cdot j \cdot q''$ with $j > i$ (region V), we have $m_1|_{q'} = m_2|_{q'}$, and from the previous lemma (6.2.1) $BV(m_2, q')$ subsumes $BV(m_1, q')$.

Now we are left with positions q' which are prefixes of q (region I and q for the case of a computation step) or $q \cdot i$ (region I, q and $q \cdot i$ for a **structure end** step).

We will show the correctness of the corresponding subterms by induction on decreasing length of q'. If $q' = q$ or $q' = q \cdot i$ respectively, it has already been proven.

For shorter q', we have

$$\begin{aligned} m_1|_{q'} = \mathsf{v}{\triangleright}v &: \mathsf{Ty}(M) \\ &:= \mathsf{Struct}\ \overline{\mathsf{v}}_1{\triangleright}\overline{v}_1 : S_1 := P_1 \ldots \mathsf{v}_j{\triangleright}v_j : \mathsf{Ty}(M_1') := m_1' \ldots \overline{\mathsf{v}}_n{\triangleright}\overline{v}_n : S_n := P_n\ \mathsf{End} \end{aligned}$$

and

$$\begin{aligned} m_2|_{q'} = \mathsf{v}{\triangleright}v &: \mathsf{Ty}(M) \\ &:= \mathsf{Struct}\ \overline{\mathsf{v}}_1{\triangleright}\overline{v}_1 : S_1 := P_1 \ldots \mathsf{v}_j{\triangleright}v_j : \mathsf{Ty}(M_2') := m_2' \ldots \overline{\mathsf{v}}_n{\triangleright}\overline{v}_n : S_n := P_n\ \mathsf{End} \end{aligned}$$

Note that M_1' and M_2' are not identical only for such q' that $q' \cdot 2 = q$ and $j = i$ and we are dealing with a **structure end** step which moved the mark to from $q \cdot i$ to $q \cdot i'$ for $i' = i + 1$.

Since m_1 is correct,

$$BV(m_1, q') \vdash \mathsf{Struct}\ \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \mathsf{v}_j \triangleright v_j : \mathsf{Ty}(M_1') := m_1' \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n := P_n\ \mathsf{End} : M$$

and by inversion

$$BV(m_1, q') \vdash \mathsf{Sig}\ \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \mathsf{v}_j \triangleright v_j : \mathsf{Ty}(M_1') \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End} <: M$$

From Lemma 6.2.1 $BV(m_2, q' \cdot 2 \cdot n') = BV(m_2, q'); \overline{v}_1 : S_1 \ldots v_i : \mathsf{Ty}(M_2') \ldots \overline{v}_n : S_n$ subsumes $BV(m_1, q' \cdot 2 \cdot n') = BV(m_1, q'); \overline{v}_1 : S_1 \ldots v_i : \mathsf{Ty}(M_1') \ldots \overline{v}_n : S_n$ where $n' = n+1$.

Since $BV(m_2, q') = BV(m_1, q')$,

$$BV(m_2, q') \ \vdash \ \begin{array}{l} \mathsf{Sig}\ \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \mathsf{v}_i \triangleright v_i : \mathsf{Ty}(M_2') \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End} \\ \ <:\ \mathsf{Sig}\ \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \mathsf{v}_i \triangleright v_i : \mathsf{Ty}(M_1') \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End} \end{array}$$

and consequently, using transitivity of subtyping, if

$$BV(m_1, q') \vdash \mathsf{Struct}\ \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \mathsf{v}_i \triangleright v_i : \mathsf{Ty}(M_1') := m_1' \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n := P_n\ \mathsf{End} : \mathsf{Ty}(M)$$

then also

$$BV(m_2, q') \vdash \mathsf{Struct}\ \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \mathsf{v}_i \triangleright v_i : \mathsf{Ty}(M_2') := m_2' \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n := P_n\ \mathsf{End} : \mathsf{Ty}(M)$$

□

Lemma 6.2.3. *Let* $m_1 = \mathsf{Struct}\ Pre_1\ Post_1\ \mathsf{End}$ *and* $m_2 = \mathsf{Struct}\ Pre_2\ Post_2\ \mathsf{End}$*. If* $\vdash m_1 : \hat{M}$ *and and* $Eval(Pre_1 \mid\mid Post_1) \longrightarrow Eval(Pre_2 \mid\mid Post_2)$ *then* $\vdash m_2 : \hat{M}$*.*

Proof. If

$$m_1 \ = \ \mathsf{Struct}\ \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n := P_n\ \mathsf{End}$$

and

$$m_2 \ = \ \mathsf{Struct}\ \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1' := P_1' \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n' := P_n'\ \mathsf{End}$$

using Lemmas 6.2.2 and 6.2.1 and the reasoning similar to the last part of the proof of the previous lemma, we can show that $\vdash m_2 : \mathsf{Sig}\ \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1' \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n'\ \mathsf{End}$ and $\vdash \mathsf{Sig}\ \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1' \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n'\ \mathsf{End} <: \mathsf{Sig}\ \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$. Hence, if $\vdash m_1 : \hat{M}$ then also $\vdash m_2 : \hat{M}$. □

6.3 Termination

In this section we prove that evaluation is terminating. Lemma 3.2.1 announced in Chapter 3 is a consequence of this fact and of the previous section.

In Section 6.3.1 we define a translation of expressions of our module system into the Weak Module Calculus of Courant [32], a calculus in which term typing is trivial and in which module reductions are terminating. Section 6.3.2 shows that the translation preserves types: the translation of a correct expression of our calculus is correct in $\mathcal{WMC}$.

Next, in Section 6.3.3 we show that **computation** steps correspond to module reductions on translated expressions and therefore every sequence of **computation** steps is finite. Unfortunately **structure end** evaluation steps imply modifications of the translated expressions which do not correspond to module reductions.

But on the other hand every evaluation step, apart from **computation**, moves the mark to the right, to the next evaluation position. Hence the rest of the termination proof aims at showing that the set of all possible evaluation positions is bounded. The bound is equal to the number of evaluation positions in the expression obtained by reducing in $\mathcal{WMC}$ the translation of the first expression in the evaluation sequence until the normal form.[1]

The only difficulty remains in showing that the bound is not increased by the modifications performed by the **structure end** step. To do this, in Sections 6.3.4 and 6.3.5 we define a certain reduction strategy in $\mathcal{WMC}$ and prove that if the reduction follows this strategy until normal forms, it is easy to see that the **structure end** step does not increase the bound.

Hence we obtain the termination of the whole evaluation process (Section 6.3.6).

6.3.1 Translation into $\mathcal{WMC}$

Below we define a translation of expressions of our syntax into the Weak Module Calculus of Courant [32]. The latter calculus is used in the work of Courant as an intermediate calculus whose role is to separate problems related with term typing from module typing. It is *weak* in a sense that typing of terms is trivial, i.e. for all terms e and t we have $e : t$. As shown in [32], module reductions (i.e. $\beta_m\delta$-reductions) of expressions well-typed in $\mathcal{WMC}$ are terminating (Theorem 5.4.10).

The definition presented below translates most classes of expressions from our calculus into $\mathcal{WMC}$. Since we are only interested in module reductions, we get rid of terms and blocks during the translation. We replace term and block components in environments, signatures and structures by (sequences) of trivial term components $v : \mathsf{Eq}(\star : \star)$ (which are correct in $\mathcal{WMC}$). Apart from this, environments and module types are translated as is, and the translation of structures (and consequently structure components) depends on the position of the mark.

Strictly formally what we define for module expressions and structure components are

[1] In fact, for the sake of simplicity, we use a bit greater bound.

three functions: when the mark is before the translated element, after that element, and inside that element. In the latter case the output of the function depends on the exact position of the mark. To simplify the presentation, we write $W(\ ||\ m)$ and $W(m\ ||\ )$ for first two cases and something like $W(\mathsf{Struct}\ \mathit{Pre}\ ||\ \mathit{Post}\ \mathsf{End})$ in the latter case. In the formulation of lemmas we often write "$\ldots W(m)$ ... *wherever the mark is in* m" to mean that the result holds for all three translating functions and for all positions of the mark in case of the third function.

Definition 6.3.1. *The translation* $W(\cdot)$ *is defined as follows:*

- *structure component sequence* $StrCs$

  $W(\overline{\mathsf{v}}\triangleright\overline{v} : S := P;\ \mathit{Pre}\ ||\ \mathit{Post}) =$
  $\quad W(\overline{\mathsf{v}}\triangleright\overline{v} : S := P\ ||\ );\ W(\mathit{Pre}\ ||\ \mathit{Post})$

  $W(\mathsf{v}\triangleright v : S := \mathsf{Struct}\ \mathit{Pre}\ ||\ \mathit{Post}\ \mathsf{End};\ StrCs) =$
  $\quad W(\mathsf{v}\triangleright v : S := \mathsf{Struct}\ \mathit{Pre}\ ||\ \mathit{Post}\ \mathsf{End});\ W(\ ||\ StrCs)$

  $W(\ ||\ \overline{\mathsf{v}}\triangleright\overline{v} : S := P;\ StrCs) =$
  $\quad W(\ ||\ \overline{\mathsf{v}}\triangleright\overline{v} : S := P);\ W(\ ||\ StrCs)$

- *structure component*

  - *abstract module*

    $W(\mathsf{v}\triangleright v : \mathsf{Ty}(M) := m\ ||\ ) =$
    $\quad \mathsf{v}\triangleright v : \mathsf{Eq}(m' : M') := m'$
    $\quad$ *where* $m' = W(m\ ||\ )$
    $\quad$ *and* $M' = SIG(m')$ *if* m *is a structure*
    $\quad$ *and* $M' = W(M)$ *otherwise*

    $W(\mathsf{v}\triangleright v : \mathsf{Ty}(M) := \mathsf{Struct}\ \mathit{Pre}\ ||\ \mathit{Post}\ \mathsf{End}) =$
    $\quad \mathsf{v}\triangleright v : \mathsf{Eq}(m' : W(M)) := m'$
    $\quad$ *where* $m' = W(\mathsf{Struct}\ \mathit{Pre}\ ||\ \mathit{Post}\ \mathsf{End})$.

    $W(\ ||\ \mathsf{v}\triangleright v : \mathsf{Ty}(M) := m) =$
    $\quad \mathsf{v}\triangleright v : \mathsf{Eq}(m' : W(M)) := m'$
    $\quad$ *where* $m' = W(\ ||\ m)$

  - *manifest module (after or before the mark)*

    $W(\mathsf{v}\triangleright v : \mathsf{Eq}(p : M) := p') =$
    $\quad \mathsf{v}\triangleright v : \mathsf{Eq}(p : W(M)) := p'$

  - *block or term (after or before the mark)*

    $W(\overline{\mathsf{v}}\triangleright\overline{v} : S := P) =$
    $\quad \mathsf{v}^1\triangleright v^1 : \mathsf{Eq}(\star : \star) := \star \ldots \mathsf{v}^n\triangleright v^n : \mathsf{Eq}(\star : \star) := \star$ *where* $\overline{v} = v^1 \ldots v^n$

- *module expression*

  $W(\mathsf{Struct}\ StrCs\ \mathsf{End}\ ||\ ) =$
  $\quad \mathsf{Struct}\ W(StrCs\ ||\ )\ \mathsf{End}$

$W(\mathsf{Struct}\ Pre \mid\mid Post\ \mathsf{End}) =$
$\quad \mathsf{Struct}\ W(Pre \mid\mid Post)\ \mathsf{End}$

$W(\ \mid\mid \mathsf{Struct}\ StrCs\ \mathsf{End}) =$
$\quad \mathsf{Struct}\ W(\ \mid\mid StrCs)\ \mathsf{End}$

$W(\mathsf{Functor}[v : M]m \mid\mid\ ) = W(\ \mid\mid \mathsf{Functor}[v : M]m) =$
$\quad \mathsf{Functor}[v : W(M)]W(\ \mid\mid m)$

$W(p \mid\mid\ ) = W(\ \mid\mid p) =$
$\quad p$

$W(\ \mid\mid p_1\, p_2) = W(p_1\, p_2 \mid\mid\ ) =$
$\quad p_1\, p_2$

- *module type*

  $W(\mathsf{Sig}\ SigCs\ \mathsf{End}) =$
  $\quad \mathsf{Sig}\ W(SigCs)\ \mathsf{End}$

  $W(\mathsf{Funsig}(v : M)M') =$
  $\quad \mathsf{Funsig}(v : W(M))W(M')$

- *signature component sequence SigCs*

  $W(v_1 : S_1 \ldots v_n : S_n) =$
  $\quad W(v_1 : S_1) \ldots W(v_n : S_n)$

- *signature component*

  $W(\mathsf{v}{\triangleright}v : \mathsf{Ty}(M))) = \mathsf{v}{\triangleright}v : \mathsf{Ty}(W(M))$

  $W(\mathsf{v}{\triangleright}v : \mathsf{Eq}(p : M)) = \mathsf{v}{\triangleright}v : \mathsf{Eq}(p : W(M))$

  $W(\overline{\mathsf{v}}{\triangleright}\overline{v} : S := P) = \mathsf{v}^1{\triangleright}v^1 : \mathsf{Eq}(\star : \star); \ldots ; \mathsf{v}^n{\triangleright}v^n : \mathsf{Eq}(\star : \star)$ *where* $\overline{v} = v^1 \ldots v^n$

- *specification*

  $W(\mathsf{Ty}(M)) =$
  $\quad \mathsf{Ty}(W(M))$

  $W(\mathsf{Eq}(p : M)) =$
  $\quad \mathsf{Eq}(p : W(M))$

  $W(S) =$
  $\quad \mathsf{Eq}(\star : \star) \ldots \mathsf{Eq}(\star : \star)$ *where the lengths of S and of the resulting sequence are equal, if S is not a module specification.*

It is easy to see that given a marked module expression m and an evaluation position q in m, we have:

- $W(m)|_q = W(\ \mid\mid m|_q)$ if the mark is before the position q,
- $W(m)|_q = W(m|_q)$ if q is a prefix of the position of the mark,
- $W(m)|_q = W(m|_q \mid\mid\ )$ otherwise

6.3.2 Correctness of the translation

We will now prove that the translation of expressions well-typed in $\vdash$ yields expressions which are well-typed in $\vdash_W$. We start by proving some simple properties, then by induction on the derivation we show correctness of the translation of module types and specifications and we conclude by showing that translating structures is correct.

Lemma 6.3.1. *For all specifications S and module types M, $W(S)\{v \mapsto p\} = W(S\{v \mapsto p\})$ and $W(M)\{v \mapsto p\} = W(M\{v \mapsto p\})$. Moreover for any module expressions m $W(m)\{v \mapsto p\} = W(m\{v \mapsto p\})$ wherever the mark is in m.*

Proof. By mutual induction on the structure of S and M respectively and by induction on the structure of m. □

Note that the definition of strengthening in [32] differs slightly from ours. Indeed, in our calculus

$$\mathsf{Funsig}(v : M)M'/p = \mathsf{Funsig}(v : M)M'$$

while in [32], we have:

$$\mathsf{Funsig}(v : M)M'/p = \mathsf{Funsig}(v : M)(M'/(p\ v))$$

Lemma 6.3.2. *Let E be an environment. For every path p, module type M and specification S,*

- *if $E \vdash p : M$ and $W(E) \vdash_W p : W(M)$ then $W(E) \vdash_W W(M/p) : \texttt{modtype}$ and $W(E) \vdash_W W(M)/p <: W(M/p)$,*
- *if $E \vdash p : S$ and $W(E) \vdash_W p : W(S)$ then $W(E) \vdash_W W(S/p) : \texttt{spec}$ and $W(E); v : W(S)/p \vdash_W v : W(S/p)$.*

Proof. By induction on the structure of M and S respectively, similarly to Lemmas 5.2.8 and 5.2.9 from [32] □

Lemma 6.3.3. *The translation of path, module type, specification and environment judgments is correct, i.e.*

- *if $E \vdash p : M$ then $W(E) \vdash_W W(p) : W(M)$,*
- *if $E \vdash p : S$ for a module specification S then $W(E) \vdash_W W(p) : W(S)$,*
- *if $E \vdash p_1 \approx p_2$ for module paths p_1 and p_2 then $W(E) \vdash_W W(p_1) \approx W(p_2)$,*
- *if $E \vdash M : \texttt{modtype}$ then $W(E) \vdash_W W(M) : \texttt{modtype}$,*
- *if $E \vdash M_1 <: M_2$ then $W(E) \vdash_W W(M_1) <: W(M_2)$,*
- *if $E \vdash \texttt{ok}$ then $W(E) \vdash_W \texttt{ok}$,*

- *if* $E \vdash S : \texttt{spec}$ *then* $W(E) \vdash_W S^i : \texttt{spec}$ *for each* S^i *in* $W(S)$.

Proof. Induction on the derivation of $E \vdash J$:

For **(Env/Empty)** it is obvious.

For **(Env/Insert)** we get $W(E; \overline{v} : S) = W(E); W(\overline{v} : S)$, and $W(E; \overline{v} : S) \vdash_W \texttt{ok}$ by induction hypothesis and application of the rule [/ENV/VAR].

For **(Env/Lookup)**, we get $W(E_1; \overline{v} : S; E_2) \vdash_W \texttt{ok}$ by induction hypothesis and if $\overline{v} : S$ is a module judgment $v : S$, then $W(E_1; v : S; E_2) = W(E_1); v : W(S); W(E_2)$ and $W(E_1; v : S; E_2) \vdash_W v : W(S)$ by [/S/VAR].

For **(Sig/Form)**, by induction hypothesis: $W(E); W(\overline{v}_1 : S_1); \ldots; W(\overline{v}_n : S_n) \vdash_W \texttt{ok}$ and hence $W(E) \vdash_W \mathsf{Sig}\ W(\overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1); \ldots; W(\overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n)\ \mathsf{End} : \texttt{modtype}$ by [M/MT/EMPTY] and successive applications of [M/MT/DECL].

For **(Sig/Access)**, since $W(p.\mathtt{v}_i) = p.\mathtt{v}_i$, by induction hypothesis we have

$$W(E) \vdash_W p : W(\mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End})$$

Since

$$W(\mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}) \ = \ \mathsf{Sig}\ W(\overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1) \ldots W(\overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n)\ \mathsf{End}$$

and since $p.\mathtt{v}_i$ is a module path, then $W(\mathtt{v}_i \triangleright v_i : S_i) = \mathtt{v}_i \triangleright v_i : W(S_i)$ and by [/S/ACCESS] in $\mathcal{WMC}$, we get $W(E) \vdash_W p.\mathtt{v}_i : W(S_i)\{\overline{v}_j \mapsto \overline{p.\mathtt{v}_j}\}$. Since the translation commutes with path substitutions, we get the desired conclusion $W(E) \vdash_W p.\mathtt{v}_i : W(S_i\{\overline{v}_j \mapsto \overline{p.\mathtt{v}_j}\})$.

For **(Sig/Sub)**, we get

$$W(E) \vdash_W \mathsf{Sig}\ W(\overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1) \ldots W(\overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n)\ \mathsf{End} : \texttt{modtype}$$

and

$$W(E) \vdash_W \mathsf{Sig}\ W(\overline{\mathtt{v}}'_1 \triangleright \overline{v}'_1 : S'_1) \ldots W(\overline{\mathtt{v}}'_{n'} \triangleright \overline{v}'_{n'} : S'_{n'})\ \mathsf{End} : \texttt{modtype}$$

from induction hypothesis. For term components, we have

$$W(E); W(\overline{v}_1 : S_1) \ldots W(\overline{v}_n : S_n) \vdash_W v'^j_i : \mathsf{Eq}(\star : \star)$$

because that is the translation of the specification of v'_i in the environment. For module components, we have

$$W(E); W(\overline{v}_1 : S_1) \ldots W(\overline{v}_n : S_n) \vdash_W v'_i : W(S'_i)$$

by induction hypothesis about paths. So, applying [M/SUB/SIG] in $\mathcal{WMC}$, we get

$$W(E) \ \vdash_W \begin{array}{l} \mathsf{Sig}\ W(\overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1) \ldots W(\overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n)\ \mathsf{End} \\ <: \ \mathsf{Sig}\ W(\overline{\mathtt{v}}'_1 \triangleright \overline{v}'_1 : S'_1) \ldots W(\overline{\mathtt{v}}'_{n'} \triangleright \overline{v}'_{n'} : S'_{n'})\ \mathsf{End} \end{array}$$

which is the desired conclusion.

The rules **(Funsig/Form)** and **(Funsig/Sub)** are obvious by induction hypothesis and corresponding rules [M/MT/PROD] and [M/SUB/PROD] of $\mathcal{WMC}$.

The **(Subsumption)** rule (where m is a path) follows from the induction hypothesis and [M/T/SUB].

The **(Strengthening)** rule follows from the induction hypothesis, Lemma 6.3.2 and [M/T/STR].

For **(Ty/Form)**, if $\mathsf{Ty}(\Psi)$ is a module specification, then $W(\mathsf{Ty}(\Psi))$ is a sequence of $\mathsf{Eq}(\star : \star)$, which are correct specifications in $\mathcal{WMC}$, by the rules [T/T/AXIOM], [T/T/TERM] and [/WS/MAN]. If $\mathsf{Ty}(\Psi)$ is a module specification, then Ψ is a module type M and Ξ is $\mathtt{modtype}$. By induction hypothesis $W(E) \vdash_W W(M) : \mathtt{modtype}$ and hence $W(E) \vdash_W \mathsf{Ty}(W(M)) : \mathtt{spec}$.

Rules **(Ty/Elim)** and **(Ty/Sat)** for module specifications follow from the induction hypothesis and rules [/T/ABS] and [/S/ABS].

For **(Eq/Form)**, again if $\mathsf{Eq}(\phi : \Psi)$ is not a module specification, than the translation yields a sequence of $\mathsf{Eq}(\star : \star)$, which are all correct in $\mathcal{WMC}$.

If $\mathsf{Eq}(\phi : \Psi)$ is a module specification $\mathsf{Eq}(p : M)$, then $W(E) \vdash_W p : W(M)$ by induction hypothesis and $W(E) \vdash_W W(M) : \mathtt{modtype}$. Hence $W(E) \vdash_W \mathsf{Eq}(p : W(M)) : \mathtt{spec}$ by [/WS/MAN] which is the desired conclusion.

The rules **(Eq/Sat)**, **(Eq/Type)** and **(Eq/Comp)** for module specifications follow easily by induction hypothesis and rules [/S/MAN], [/T/MAN] and [/E/SPEC].

Equality rules on module paths: **(Conv/Refl)**, **(Conv/Trans)**, **(Conv/Symm)** and **(Conv/Path)** follow from induction hypothesis and rules [/E/MU], [/E/TRANS], [/E/SYM] and [/E/ACCESS], respectively. $\square$

We say that a module expression m is a partially evaluated marked module expression if m is evaluated up to the mark. In other words, if $m = \mathsf{Struct}\ \mathit{Pre} \mid\mid \mathit{Post}\ \mathsf{End}$ then Pre is evaluated and if the mark is after m then the whole module expression m is evaluated and in particular it cannot be a path application.

Lemma 6.3.4. *Let m be a partially evaluated marked module expression. If $E \vdash m : M$ then $W(E) \vdash_W W(m) : W(M)$, wherever the mark is in m. Moreover if m is a structure* $\mathsf{Struct}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n := P_n\ \mathsf{End}$ *then*

$$
\begin{array}{rcl}
W(m) & = & \mathsf{Struct} \\
 & & \quad \mathtt{v}_1^1 \triangleright v_1^1 : S_1^{1\prime} := P_1^{1\prime} \ldots \mathtt{v}_1^{k_1} \triangleright v_1^{k_1} : S_1^{k_1\prime} := P_1^{k_1\prime} \\
 & & \quad \ldots \\
 & & \quad \mathtt{v}_n^1 \triangleright v_n^1 : S_n^{1\prime} := P_n^{1\prime} \ldots \mathtt{v}_n^{k_n} \triangleright v_n^{k_n} : S_n^{k_n\prime} := P_n^{k_n\prime} \\
 & & \mathsf{End}
\end{array}
$$

where $v_1^1 \ldots v_1^{k_1} = \overline{v}_1, \ldots, v_n^1 \ldots v_n^{k_n} = \overline{v}_n$ and

$$W(E) \vdash_W SIG(W(m)) <: W(SIG(m))$$

Proof. By induction on the structure of m, using the previous lemma.

- If m is a structure, then by inversion $E \vdash SIG(m) <: M$ and by Lemma 6.3.3, $W(E) \vdash_W W(SIG(m)) <: W(M)$.

  Now we show by induction on i that for every $i = 1 \ldots n$ and every $j = 1 \ldots k_i$

$$E_{i-1} \vdash_W P_i^{j\prime} : S_i^{j\prime} \quad \text{and} \quad E_i \vdash_W v_i^j : \hat{S}_i^j$$

  where $E_l = W(E); v_1^1 : S_1^{1\prime} \ldots v_1^{k_1} : S_1^{k_1\prime} \ \ldots\ v_l^1 : S_l^{1\prime} \ldots v_l^{k_l} : S_l^{k_l\prime}$ and $\hat{S}_i^1 \ldots \hat{S}_i^{k_i} = W(S_i)$

  - If $\mathsf{v}_i \triangleright v_i : S_i := P_i$ is not a module component, then the resulting fragment of the structure $W(m)$ is

$$\mathsf{v}_i^1 \triangleright v_i^1 : \mathsf{Eq}(\star : \star) := \star; \ldots; \mathsf{v}_i^{k_i} \triangleright v_i^{k_i} : \mathsf{Eq}(\star : \star) := \star$$

    and $\hat{S}_i^j = \mathsf{Eq}(\star : \star)$ for all j. Hence the second judgment holds trivially and the first follows from the fact that in any environment $\hat{E}$, we have $\hat{E} \vdash_W \star : \mathsf{Eq}(\star : \star)$
  - If $\mathsf{v}_i \triangleright v_i : S_i := P_i$ is a module judgment, then $k_i = 1$. If P_i is a structure m', then we consider two cases, depending on the position of the mark in m. If the mark is after i, then according to the definition of the translation function, $S_i = \mathsf{Ty}(SIG(m'))$, $P_i^{1\prime} = W(m' \mid\mid\ )$ and $S_i^{1\prime} = \mathsf{Eq}(W(m' \mid\mid\ ) : SIG(W(m' \mid\mid\ )))$. Since

$$E; \overline{v}_1 : S_1; \ldots; \overline{v}_{i-1} : S_{i-1} \vdash m' : SIG(m')$$

    by big induction hypothesis we have

$$W(E); W(\overline{v}_1 : S_1); \ldots; W(\overline{v}_{i-1} : S_{i-1}) \ \vdash_W \ W(m' \mid\mid\ ) : W(SIG(m'))$$

    and

$$W(E); W(\overline{v}_1 : S_1); \ldots; W(\overline{v}_{i-1} : S_{i-1}) \ \vdash_W \ SIG(W(m' \mid\mid\ )) <: W(SIG(m'))$$

    Since $W(m' \mid\mid\ )$ is typable, it can also be typed by its primary signature $SIG(W(m' \mid\mid\ ))$. Using the second part of the small induction hypothesis for smaller i we can show that E_{i-1} subsumes $W(E); W(\overline{v}_1 : S_1); \ldots; W(\overline{v}_{i-1} : S_{i-1})$. By weakening lemma for $\mathcal{WMC}$ we get:

$$E_{i-1} \ \vdash_W \ W(m' \mid\mid\ ) : SIG(W(m' \mid\mid\ ))$$

    and

$$E_{i-1} \ \vdash_W \ SIG(W(m' \mid\mid\ )) <: W(SIG(m'))$$

    From the latter two judgments, using subsumption and specification rules we get

$$E_{i-1} \ \vdash_W \ W(m' \mid\mid\ ) \ : \ \mathsf{Eq}(W(m' \mid\mid\ ) : SIG(W(m' \mid\mid\ )))$$

and

$$E_{i-1}; v_i : \mathsf{Eq}(W(m' \mid\mid) : SIG(W(m' \mid\mid))) \vdash_W \; v_i : \mathsf{Ty}(W(SIG(m')))$$

which are the two facts that we have to prove, since $W(S_i) = \mathsf{Ty}(W(SIG(m')))$.

If the mark in m is inside or after i, then $S_i = \mathsf{Ty}(M')$ for some module type M' and $S_i^{1'} = \mathsf{Eq}(W(m') : \mathsf{Ty}(W(M'))$. By the big induction hypothesis, we get

$$W(E); W(\overline{v}_1 : S_1); \ldots ; W(\overline{v}_{i-1} : S_{i-1}) \vdash_W \; W(m') : W(M')$$

and by weakening and specification satisfaction

$$E_{i-1} \vdash_W \; W(m') : \mathsf{Eq}(W(m) : W(M'))$$

which is the first inductive fact that we have to prove, as $P_i^1 = W(m')$. The other is very simple since $W(S_i) = \mathsf{Ty}(W(M'))$ and obviously

$$E_{i-1}; v_i : \mathsf{Eq}(W(m') : W(M')) \vdash_W \; v_i : \mathsf{Ty}(W(M'))$$

– If $\mathsf{v}{\triangleright}v : S_i := P_i$ is a module component where S_i is an abstract specification and P_i is not a structure, then we proceed exactly like in the second case for structures.
– If $\mathsf{v}{\triangleright}v : S_i := P_i$ is a module component where S_i is a manifest specification, then $S_i = \mathsf{Eq}(p : M')$, $P_i = p'$ and $S_i^{1'} = \mathsf{Eq}(p : W(M'))$ and $P_i^{1'} = p'$. Since $W(S_i) = S_i^{1'} = \mathsf{Eq}(p : W(M'))$ we can simply use Lemma 6.3.3 to get

$$W(E); W(\overline{v}_1 : S_1); \ldots ; W(\overline{v}_{i-1} : S_{i-1}) \vdash_W \; p' : \mathsf{Eq}(p : W(M'))$$

and weakening to get $E_{i-1} \vdash_W \; p' : \mathsf{Eq}(p : W(M'))$. The second inductive fact is trivial.

Let us summarize the case where m is a structure. From first inductive facts we get the correctness of all implementations of the structure $W(m)$, which gives us

$$W(E) \vdash_W W(m) : SIG(W(m))$$

From the second inductive facts, we have

$$W(E) \vdash_W SIG(W(m)) <: \mathsf{Sig}\; W(\overline{\mathsf{v}}_1{\triangleright}\overline{v}_1 : S_1) \ldots W(\overline{\mathsf{v}}_n{\triangleright}\overline{v}_n : S_n)\; \mathsf{End}$$

which is $W(E) \vdash_W SIG(W(m)) <: W(SIG(m))$.

Using subsumption and the fact that $W(E) \vdash_W W(SIG(m)) <: W(M)$ following from inversion and Lemma 6.3.3, we get the desired conclusion $W(E) \vdash_W W(m) : W(M)$.

- If m is of the form $\mathsf{Functor}(\hat{v} : M')m'$ then regardless of the position of the mark, its translation is always $\mathsf{Functor}(\hat{v} : W(M'))W(\ || \ m')$. So if $E \vdash \mathsf{Functor}[\hat{v} : M']m' : M$, then by inversion we have $E \vdash \mathsf{Functor}[\hat{v} : M']m' : \mathsf{Funsig}(\hat{v} : M')M''$ and $E \vdash \mathsf{Funsig}(\hat{v} : M')M'' <: M$. Again, by inversion $E; \hat{v} : \mathsf{Ty}(M') \vdash m' : M''$. Using Lemma 6.3.3, we have

$$W(E) \vdash_W W(\mathsf{Funsig}(\hat{v} : M')M'') <: W(M)$$

By induction hypothesis, $W(E); \hat{v} : W(M') \vdash_W W(\ || \ m) : W(M'')$ and hence $W(E) \vdash_W \mathsf{Functor}[\hat{v} : W(M')]W(\ || \ m) : \mathsf{Funsig}(\hat{v} : W(M'))W(M'')$, which is equivalent to $W(E) \vdash_W W(\mathsf{Functor}[\hat{v} : M']m) : W(\mathsf{Funsig}(\hat{v} : M')M'')$. By subsumption, we obtain $W(E) \vdash_W W(\mathsf{Functor}[\hat{v} : M']m) : W(M)$.

- If m is a path p then we can immediately apply Lemma 6.3.3.

- If m is an application of paths $p_1\, p_2$, then by inversion we get the following judgments: $E \vdash p_1 : \mathsf{Funsig}(\hat{v} : M')M''$, $E \vdash p_2 : M'$ and $E \vdash M''\{\hat{v} \mapsto p_2\} <: M$. We can apply Lemma 6.3.3 to all of them and get $W(E) \vdash_W p_1 : W(\mathsf{Funsig}(\hat{v} : M')M'')$, $W(E) \vdash_W p_2 : W(M')$ and $W(E) \vdash_P W(M''\{\hat{v} \mapsto p_2\}) <: W(M)$.

  Since the following equalities hold

$$\begin{aligned} W(\mathsf{Funsig}(\hat{v} : M')M'') &= \mathsf{Funsig}(\hat{v} : W(M'))W(M'') \\ W(M''\{\hat{v} \mapsto p_2\}) &= W(M'')\{\hat{v} \mapsto p_2\} \\ W(p_1\, p_2) &= p_1\, p_2 \end{aligned}$$

  we can use the application and subsumption to get the desired conclusion

$$W(E) \vdash_W p_1\, p_2 : W(M)$$

□

Corollary 6.3.5. *Let* $m = \mathsf{Struct}\ \mathit{Pre} \ || \ \mathit{Post}\ \mathsf{End}$ *be a partially evaluated marked structure. If* $\vdash m : M$ *then* $\vdash_W W(\mathsf{Struct}\ \mathit{Pre} \ || \ \mathit{Post}\ \mathsf{End}) : W(M)$

6.3.3 Computation is a delta-mu reduction

Given a **computation** step evaluating a module path application $p_1\, p_2$, the first lemma in this section shows that in the translated term p_1 δ-reduces to the translation of its value $val_{p_1}(\mathit{Pre})$. The second lemma shows that the **computation** step corresponds to a non-empty sequence of $\mu\delta$-reductions between the translated structures.

Lemma 6.3.6. *Let* $\vdash \mathsf{Struct}\ \mathit{Pre} \ || \ \mathit{Post}\ \mathsf{End} : \hat{M}$ *where* q *is a position of the mark and* Pre *is evaluated. If* $val_{p'}(\mathit{Pre}) = m$ *then*

$$BV(W(\mathsf{Struct}\ \mathit{Pre} \ || \ \mathit{Post}\ \mathsf{End}), q)) \vdash_W p' \longrightarrow_\delta W(m \ || \ )$$

Moreover if m *is a structure, then*

$$BV(W(\mathsf{Struct}\ \mathit{Pre} \ || \ \mathit{Post}\ \mathsf{End}), q)) \vdash_\delta p' : \mathsf{Eq}(W(m \ || \ ) : SIG(W(m \ || \ )))$$

Proof. By induction on the structure of the path p'.

- If p' is a variable v, then either $Pre = Pre_1\ \mathtt{v}{\rhd}v : \mathsf{Eq}(p : M) := m\ Pre_2$ and $val_v(Pre) = p$ or $Pre = Pre_1\ \mathtt{v}{\rhd}v : \mathsf{Ty}(M) := m\ Pre_2$ and $val_v(Pre) = m$. In the first case $m = \hat{p}$ for some $\hat{p}$ and we have

$$BV(W(\mathsf{Struct}\ Pre_1\ \mathtt{v}{\rhd}v : \mathsf{Eq}(p : M) := m\ Pre_2 \mid\mid Post\ \mathsf{End}, q) \\ = E_1; v : \mathsf{Eq}(p : M'); E_2$$

  for some E_1, E_2 and M' and hence

$$BV(W(\mathsf{Struct}\ Pre \mid\mid Post\ \mathsf{End}, q) \vdash_W v \longrightarrow_\delta p$$

  In the second case

$$BV(W(\mathsf{Struct}\ Pre_1\ \mathtt{v}{\rhd}v : \mathsf{Ty}(M) := m\ Pre_2 \mid\mid Post\ \mathsf{End}, q) \\ = E_1; v : \mathsf{Eq}(W(m \mid\mid) : M'); E_2$$

  for some E_1, E_2 and M'. Consequently,

$$E_1; v : \mathsf{Eq}(W(m \mid\mid) : M'); E_2 \vdash_\delta v : \mathsf{Eq}(W(m \mid\mid) : M')$$

  If m is a structure then by definition of $W(\mathtt{v}{\rhd}v : \mathsf{Ty}(M) := m \mid\mid)$, $M' = SIG(W(m \mid\mid))$.

- If $p' = p.\mathtt{v}_i$ and $val_p(Pre) = p''$ then $val_{p.\mathtt{v}_i}(Pre) = p''.\mathtt{v}_i$
  By induction hypothesis: $BV(W(\mathsf{Struct}\ Pre \mid\mid Post\ \mathsf{End}), q)) \vdash_W p \longrightarrow_\delta p''$ because $p'' = W(p'' \mid\mid)$. Hence $BV(W(\mathsf{Struct}\ Pre \mid\mid Post\ \mathsf{End}), q)) \vdash_W p.\mathtt{v}_i \longrightarrow_\delta p''.\mathtt{v}_i$.

- If $p' = p.\mathtt{v}_i$ and $val_p(Pre) = m$ where

$$m \;=\; \mathsf{Struct}\ \overline{\mathtt{v}}_1{\rhd}\overline{v}_1 : S_1 := P_1 \ldots \mathtt{v}_i{\rhd}v_i : S_i := m_i \ldots \overline{\mathtt{v}}_n{\rhd}\overline{v}_n : S_n := P_n\ \mathsf{End}$$

  then either $S_i = \mathsf{Eq}(p'' : M)$ and $val_{p.\mathtt{v}_i}(Pre) = p''\delta$, where $\delta = \{\overline{v}_j \mapsto \overline{p.\mathtt{v}_j}\}$ or $S_i = \mathsf{Ty}(M)$ and $val_{p.\mathtt{v}_i}(Pre) = m_i\delta$.
  By definition of $W(\cdot)$,

$$\begin{array}{rcl} W(m \mid\mid) & = & \mathsf{Struct} \\ & & \quad \mathtt{v}_1^1{\rhd}v_1^1 : S_1^{1\prime} := P_1^{1\prime} \ldots \mathtt{v}_1^{k_1}{\rhd}v_1^{k_1} : S_1^{k_1\prime} := P_1^{k_1\prime} \\ & & \quad \ldots \\ & & \quad \mathtt{v}_i{\rhd}v_i : S_i' := m_i' \\ & & \quad \ldots \\ & & \quad \mathtt{v}_n^1{\rhd}v_n^1 : S_n^{1\prime} := P_n^{1\prime} \ldots \mathtt{v}_n^{k_n}{\rhd}v_n^{k_n} : S_n^{k_n\prime} := P_n^{k_n\prime} \\ & & \mathsf{End} \end{array}$$

By induction hypothesis

$$BV(W(\mathsf{Struct}\ Pre \mid\mid Post\ \mathsf{End}), q)) \ \vdash_\delta \ p : \mathsf{Eq}(W(m \mid\mid \,) : SIG(W(m \mid\mid \,)))$$

and hence

$$BV(W(\mathsf{Struct}\ Pre \mid\mid Post\ \mathsf{End}), q)) \vdash_\delta p.\mathtt{v}_i : S'_i\delta$$

If $S_i = \mathsf{Eq}(p'' : M)$ then $S'_i = \mathsf{Eq}(p'' : W(M))$ and hence

$$\begin{aligned}
&BV(W(\mathsf{Struct}\ Pre \mid\mid Post\ \mathsf{End}), q)) \vdash_\delta p.\mathtt{v}_i : \mathsf{Eq}(p''\delta : W(M)\delta)\\
&BV(W(\mathsf{Struct}\ Pre \mid\mid Post\ \mathsf{End}), q)) \vdash_W p.\mathtt{v}_i \longrightarrow_\delta p''\delta
\end{aligned}$$

Otherwise $S'_i = \mathsf{Eq}(m'_i : M')$, with $m'_i = W(m_i \mid\mid \,)$ and $M' = SIG(m'_i)$ if m_i is a structure. Consequently

$$\begin{aligned}
&BV(W(\mathsf{Struct}\ Pre \mid\mid Post\ \mathsf{End}), q)) \vdash_\delta p.\mathtt{v}_i : \mathsf{Eq}(m'_i\delta : M'\delta)\\
&BV(W(\mathsf{Struct}\ Pre \mid\mid Post\ \mathsf{End}), q)) \vdash_W p.\mathtt{v}_i \longrightarrow_\delta m'_i\delta
\end{aligned}$$

By Lemma 6.3.1 $W(m_i\delta \mid\mid \,) = W(m_i \mid\mid \,)\delta$, so the first part of the lemma is proved.

If m_i is a structure, so are m'_i and $m'_i\delta$. Hence, since $M' = SIG(m'_i)$ then also $M'\delta = SIG(m'_i)\delta = SIG(m'_i\delta)$, which concludes the proof of the second part.

□

Lemma 6.3.7. *Let* $\vdash \mathsf{Struct}\ Pre_1\ Post_1\ \mathsf{End} : M$ *with* Pre_1 *evaluated. If*

$$Eval(Pre_1 \mid\mid Post_1) \longrightarrow Eval(Pre_2 \mid\mid Post_2)$$

by a **computation** *step, then*

$$\vdash_W W(\mathsf{Struct}\ Pre_1 \mid\mid Post_1\ \mathsf{End}) \longrightarrow^+_{\mu\delta} W(\mathsf{Struct}\ Pre_2 \mid\mid Post_2\ \mathsf{End})$$

Proof. Let $\hat{m}_1 = \mathsf{Struct}\ Pre_1 \mid\mid Post_1\ \mathsf{End}$ and $\hat{m}_2 = \mathsf{Struct}\ Pre_2 \mid\mid Post_2\ \mathsf{End}$ and let $m_1 = W(\hat{m}_1)$ and $m_2 = W(\hat{m}_2)$. Let q' be the position in $\hat{m}_1$ at which the **computation** step is performed.

We show by induction on the decreasing length of even prefixes q of $q' \cdot 2$ that

$$BV(m_1, q) \ \vdash_W \ m_1|_q \longrightarrow^+_{\mu\delta} m_2|_q$$

Since q' is a shape position, its length is odd, therefore the longest q is $q' \cdot 2$.

We have $\hat{m}_1|_{q'} \ = \ \mathtt{v} \triangleright v : \mathsf{Ty}(M) := p_1\, p_2$ and hence

$$\hat{m}_1|_{q'\cdot 2} = m_1|_{q'\cdot 2} = p_1\, p_2 \quad \text{ and } \quad BV(m_1, q' \cdot 2) = BV(m_1, q')$$

If $val_{p_1}(Pre_1) = p'$, then by Lemma 6.3.6, $BV(m_1, q') \vdash_W p_1 \longrightarrow_\delta p'$. Hence, we also have $BV(m_1, q' \cdot 2) \vdash_W p_1\, p_2 \longrightarrow_\delta p'\, p_2$, where $p'\, p_2 = m_2|_{q'\cdot 2} = \hat{m}_2|_{q'\cdot 2}$.

If $val_{p_1}(Pre_1) = \mathsf{Functor}[\hat{v} : M']m$, then $BV(m_1, q') \vdash_W p_1 \longrightarrow_\delta W(\mathsf{Functor}[\hat{v} : M']m \mid\mid)$. We have $\hat{m}_2|_{q'\cdot 2} = m\{\hat{v} \mapsto p_2\}$ and $m_2|_{q'\cdot 2} = W(\mid\mid m\{\hat{v} \mapsto p_2\})$. Since

$$W(\mathsf{Functor}[\hat{v} : M']m \mid\mid) = \mathsf{Functor}[\hat{v} : W(M')]W(\mid\mid m)$$

we have

$$BV(m_1, q' \cdot 2) \vdash_W p_1\, p_2 \longrightarrow_\delta \mathsf{Functor}[\hat{v} : W(M')]W(\mid\mid m)\ p_2 \longrightarrow_\mu W(\mid\mid m)\{\hat{v} \mapsto p_2\}$$

The latter expression is equal $W(\mid\mid m\{\hat{v} \mapsto p_2\})$ and hence

$$BV(m_1, q' \cdot 2) \vdash_W m_1|_{q'\cdot 2} \longrightarrow^+_{\mu\delta} m_2|_{q'\cdot 2}$$

If q is a strict even prefix of $q' \cdot 2$ then $q' \cdot 2 = q \cdot j \cdot 2 \cdot \hat{q}$ for some j and $\hat{q}$ of even length. Therefore

$$\begin{array}{rcl} m_1|_q & = & \mathsf{Struct} \\ & & \quad \mathtt{v}_1^1 \triangleright v_1^1 : {S_1^1}' := {P_1^1}' \ldots \mathtt{v}_1^{k_1} \triangleright v_1^{k_1} : {S_1^{k_1}}' := {P_1^{k_1}}' \\ & & \quad \ldots \\ & & \quad \mathtt{v}_j \triangleright v_j : \mathsf{Eq}(m_1' : W(M')) := m_1' \\ & & \quad \ldots \\ & & \quad \mathtt{v}_n^1 \triangleright v_n^1 : {S_n^1}' := {P_n^1}' \ldots \mathtt{v}_n^{k_n} \triangleright v_n^{k_n} : {S_n^{k_n}}' := {P_n^{k_n}}' \\ & & \mathsf{End} \end{array}$$

where $m_1' = W(\hat{m}_1|_{q\cdot j\cdot 2})$ and

$$\begin{array}{rcl} m_2|_q & = & \mathsf{Struct} \\ & & \quad \mathtt{v}_1^1 \triangleright v_1^1 : {S_1^1}' := {P_1^1}' \ldots \mathtt{v}_1^{k_1} \triangleright v_1^{k_1} : {S_1^{k_1}}' := {P_1^{k_1}}' \\ & & \quad \ldots \\ & & \quad \mathtt{v}_j \triangleright v_j : \mathsf{Eq}(m_2' : W(M')) := m_2' \\ & & \quad \ldots \\ & & \quad \mathtt{v}_n^1 \triangleright v_n^1 : {S_n^1}' := {P_n^1}' \ldots \mathtt{v}_n^{k_n} \triangleright v_n^{k_n} : {S_n^{k_n}}' := {P_n^{k_n}}' \\ & & \mathsf{End} \end{array}$$

where $m_2' = W(\hat{m}_2|_{q\cdot j\cdot 2})$.

Since $m_1' = m_1|_{q\cdot j\cdot 2}$ and $m_2' = m_2|_{q\cdot j\cdot 2}$, by induction hypothesis

$$BV(m_1, q \cdot j \cdot 2) \vdash_W m_1' \longrightarrow^+_{\mu\delta} m_2'$$

Since $BV(m_1, q \cdot j \cdot 2) = BV(m_1, q \cdot j \cdot 1 \cdot 1)$, we also have

$$BV(m_1, q \cdot j \cdot 1 \cdot 1) \vdash_W m_1' \longrightarrow^+_{\mu\delta} m_2'$$

and hence

$$BV(m_1, q) \vdash_W m_1|_q \longrightarrow^+_{\mu\delta} m_2|_q$$

Using this fact for $q = \epsilon$, we obtain $\vdash_W m_1 \longrightarrow^+_{\mu\delta} m_2$ which concludes the proof. $\square$

Since by theorem 5.4.10 from [32], $\longrightarrow_{\mu\delta}$ is strongly normalizing, we have the following corollary:

Corollary 6.3.8. *Every sequence of consecutive* **computation** *steps is finite.*

6.3.4 Shape positions

A **structure end** step implies modification of the module type M in a manifest specification $\mathsf{Eq}(m : M)$ in the translated expressions. In this section we define a congruence identifying expressions which differ only in this way. In the next section we will define a normalisation strategy that preserves this congruence, allowing to prove that **structure end** steps do not increase the upper bound on the number of evaluation positions.

Definition 6.3.2. *Let Ψ be an expression and q a position in Ψ. We say that q is a* shape position *if it is not of the form $q' \cdot 2 \cdot q''$ where $\Psi|_{q'} = \mathsf{Eq}(\psi : \Phi)$. We denote the set of shape positions of Ψ by $Sh(\Psi)$.*

Definition 6.3.3. *Let $\leftrightsquigarrow$ be the least congruence satisfying the condition: if $m_1 \leftrightsquigarrow m_2$ then $\mathsf{Eq}(m_1 : M_1) \leftrightsquigarrow \mathsf{Eq}(m_2 : M_2)$ for any M_1 and M_2. If $\Psi \leftrightsquigarrow \Psi'$ we say that they have* similar shape*.*

The notion of similar shape naturally extends to environments. Obviously $E \leftrightsquigarrow E'$ if $E = v_1 : S_1; \dots v_n : S_n$ and $E' = v_1 : S'_1; \dots v_n : S'_n$, and we have $S_1 \leftrightsquigarrow S'_1 \dots S_n \leftrightsquigarrow S'_n$.

Lemma 6.3.9. *If $\Psi \leftrightsquigarrow \Psi'$ and q is a shape position in Ψ, then:*

- *q is a shape position in Ψ';*
- *$\Psi|_q \leftrightsquigarrow \Psi'|_q$ and $BV(\Psi, q) \leftrightsquigarrow BV(\Psi', q)$;*
- *if $\psi \leftrightsquigarrow \psi'$ then $\Psi[\psi]_q \leftrightsquigarrow \Psi'[\psi']_q$;*
- *if $\psi \leftrightsquigarrow \psi'$ then $\Psi\{v \mapsto \psi\} \leftrightsquigarrow \Psi'\{v \mapsto \psi'\}$;*

Proof. Easy. ∎

Lemma 6.3.10. *Let $\vdash \mathsf{Struct}\ Pre_1\ Post_1\ \mathsf{End} : M$ where Pre is evaluated. If*

$$Eval(Pre_1 \mid\mid Post_1) \longrightarrow Eval(Pre_2 \mid\mid Post_2)$$

by a **structure end** *step, then*

$$W(\mathsf{Struct}\ Pre_1 \mid\mid Post_1\ \mathsf{End}) \leftrightsquigarrow W(\mathsf{Struct}\ Pre_2 \mid\mid Post_2\ \mathsf{End})$$

Proof. Let $\hat{m}_1 = \mathsf{Struct}\ Pre_1 \,||\, Post_1\ \mathsf{End}$ and $\hat{m}_2 = \mathsf{Struct}\ Pre_2 \,||\, Post_2\ \mathsf{End}$ and let $m_1 = W(\hat{m}_1)$ and $m_2 = W(\hat{m}_2)$. The **structure end** step moved the mark from the position $q' \cdot i \cdot 2 \cdot n'$ to $q' \cdot i'$ for $i' = i + 1$.

We show by induction on decreasing length of even prefixes q of q' that $m_1|_q \leftrightsquigarrow m_2|_q$. Since $q' \cdot i \cdot 2 \cdot n'$ is an evaluation position, its length is odd and the length of q' is even. We have

$$\begin{array}{rcl} \hat{m}_1|_{q'} & = & \mathsf{Struct} \\ & & \quad \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \\ & & \quad \ldots \\ & & \quad \mathsf{v}_i \triangleright v_i : \mathsf{Ty}(M) := \mathsf{Struct}\ StrCs \,||\ \mathsf{End} \\ & & \quad \ldots \\ & & \quad \overline{\mathsf{v}}_l \triangleright \overline{v}_l : S_l := P_l \\ & & \mathsf{End} \end{array}$$

and

$$\begin{array}{l} \hat{m}_2|_{q'} = \mathsf{Struct} \\ \quad \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 : P_1 \\ \quad \ldots \\ \quad \mathsf{v}_i \triangleright v_i : \mathsf{Ty}(SIG(\mathsf{Struct}\ StrCs\ \mathsf{End})) := \mathsf{Struct}\ StrCs\ \mathsf{End}\ \ || \\ \quad \ldots \\ \quad \overline{\mathsf{v}}_l \triangleright \overline{v}_l : S_l : P_l \\ \mathsf{End} \end{array}$$

On the translated side, we have

$$\begin{array}{rcl} m_1|_{q'} = W(\hat{m}_1|_{q'}) & = & \mathsf{Struct} \\ & & \quad \mathsf{v}_1^1 \triangleright v_1^1 : {S_1^1}' := {P_1^1}' \ldots \mathsf{v}_1^{k_1} \triangleright v_1^{k_1} : {S_1^{k_1}}' := {P_1^{k_1}}' \\ & & \quad \ldots \\ & & \quad \mathsf{v}_i \triangleright v_i : \mathsf{Eq}(m' : W(M)) := m' \\ & & \quad \ldots \\ & & \quad \mathsf{v}_l^1 \triangleright v_l^1 : {S_l^1}' := {P_l^1}' \ldots \mathsf{v}_l^{k_l} \triangleright v_l^{k_l} : {S_l^{k_l}}' := {P_l^{k_l}}' \\ & & \mathsf{End} \end{array}$$

and

$$\begin{array}{rcl} m_2|_{q'} = W(\hat{m}_2|_{q'}) & = & \mathsf{Struct} \\ & & \quad \mathsf{v}_1^1 \triangleright v_1^1 : {S_1^1}' := {P_1^1}' \ldots \mathsf{v}_1^{k_1} \triangleright v_1^{k_1} : {S_1^{k_1}}' := {P_1^{k_1}}' \\ & & \quad \ldots \end{array}$$

$$
\begin{array}{l}
\quad \mathsf{v}_i \triangleright v_i : \mathsf{Eq}(m' : SIG(m')) := m' \\
\quad \ldots \\
\quad \mathsf{v}_l^1 \triangleright v_l^1 : {S_l^1}' := {P_l^1}' \ldots \mathsf{v}_l^{k_l} \triangleright v_l^{k_l} : {S_l^{k_l}}' := {P_l^{k_l}}' \\
\mathsf{End}
\end{array}
$$

where $m' = W(\mathsf{Struct}\ StrCs \mid\mid\ \mathsf{End}) = W(\mathsf{Struct}\ StrCs\ \mathsf{End} \mid\mid\ )$

Both structures $m_1|_{q'}$ and $m_2|_{q'}$ differ only on position $q' \cdot i \cdot 1 \cdot 2$ which is not a shape position, therefore they have similar shape.

For shorter even prefixes q such that $q' = q \cdot j \cdot 2 \cdot \hat{q}$ for some j and $\hat{q}$, we have

$$
\begin{array}{rcl}
m_1|_q & = & \mathsf{Struct} \\
& & \quad \mathsf{v}_1^1 \triangleright v_1^1 : {S_1^1}' := {P_1^1}' \ldots \mathsf{v}_1^{k_1} \triangleright v_1^{k_1} : {S_1^{k_1}}' := {P_1^{k_1}}' \\
& & \quad \ldots \\
& & \quad \mathsf{v}_i \triangleright v_i : \mathsf{Eq}(m_1' : W(M')) := m_1' \\
& & \quad \ldots \\
& & \quad \mathsf{v}_l^1 \triangleright v_l^1 : {S_l^1}' := {P_l^1}' \ldots \mathsf{v}_l^{k_l} \triangleright v_l^{k_l} : {S_l^{k_l}}' := {P_l^{k_l}}' \\
& & \mathsf{End}
\end{array}
$$

and

$$
\begin{array}{rcl}
m_2|_q & = & \mathsf{Struct} \\
& & \quad \mathsf{v}_1^1 \triangleright v_1^1 : {S_1^1}' := {P_1^1}' \ldots \mathsf{v}_1^{k_1} \triangleright v_1^{k_1} : {S_1^{k_1}}' := {P_1^{k_1}}' \\
& & \quad \ldots \\
& & \quad \mathsf{v}_i \triangleright v_i : \mathsf{Eq}(m_2' : W(M')) := m_2' \\
& & \quad \ldots \\
& & \quad \mathsf{v}_l^1 \triangleright v_l^1 : {S_l^1}' := {P_l^1}' \ldots \mathsf{v}_l^{k_l} \triangleright v_l^{k_l} : {S_l^{k_l}}' := {P_l^{k_l}}' \\
& & \mathsf{End}
\end{array}
$$

where $m_1' = W(\hat{m}_1|_{q \cdot j \cdot 2})$ and $m_2' = W(\hat{m}_2|_{q \cdot j \cdot 2})$. The latter two have similar shape by induction hypothesis, since $m_1' = m_1|_{q \cdot j \cdot 2}$ and $m_2' = m_2|_{q \cdot j \cdot 2}$.

Consequently $m_1|_q \leftrightsquigarrow m_2|_q$.

If q is ϵ, we get $m_1 \leftrightsquigarrow m_2$ which concludes the proof. $\square$

6.3.5 Delta-force reduction

We present here a strategy for the delta-reduction, called delta-force reduction, where the evaluated delta-redex is the smallest possible. In other words a delta-derivation of a delta-force redex does not allow to delta reduce any other redex.

The normalisation strategy announced in the previous section consists in evaluating all $\mu\delta_F$-redexes on shape positions first. Lemma 6.3.12 shows that given two expressions

m_1 and m_2 such that $m_1 \leftrightsquigarrow m_2$, the $\mu\delta_F$-reductions on shape positions in m_1 may be simulated in m_2 in such a way that the equivalence is preserved, i.e. after every reduction step we have $m_1^i \leftrightsquigarrow m_2^i$. Once all such redexes are reduced, there are no more $\mu\delta$-redexes on shape positions. In the next section we prove that further $\mu\delta$-normalisation preserves the equivalence as well (Lemma 6.3.14).

Definition 6.3.4. *Let* delta-force *reduction* $\longrightarrow_{\delta_F}$ *be defined as the smallest congruence such that* $E \vdash_W \phi \longrightarrow_{\delta_F} \phi'$ *if* $E \vdash_{\delta_F} \phi : \mathsf{Eq}(\phi' : \Psi)$ *is derivable in the following inference system:*

[/S/ACCESS]

$$\frac{E \vdash_{\delta_F} m : \mathsf{Sig}\ \mathsf{v}_1 \triangleright v_1 : S_1 \ldots \mathsf{v}_n \triangleright v_n : S_n\ \mathsf{End}}{E \vdash_{\delta_F} m.\mathsf{v}_i : S_i\{v_j \mapsto m.\mathsf{v}_j\}_{j=1\ldots i-1}}$$

[/S/VAR]

$$\frac{}{E; v : S; E' \vdash_{\delta_F} v : S}$$

[/S/ABS]

$$\frac{E \vdash_{\delta_F} m : M}{E \vdash_{\delta_F} m : \mathsf{Ty}(M)} \text{ if } m \text{ is not a variable or a selection}$$

[/T/ABS]

$$\frac{E \vdash_{\delta_F} m : \mathsf{Ty}(M)}{E \vdash_{\delta_F} m : M} \text{ if } m \text{ is a variable or a selection}$$

[M/T/APP]

$$\frac{E \vdash_{\delta_F} m_1 : \mathsf{Funsig}(v : M)M'}{E \vdash_{\delta_F} (m_1\ m_2) : M'\{v \mapsto m_2\}}$$

[M/T/LAM]

$$\frac{E; v : \mathsf{Ty}(M) \vdash_{\delta_F} m : M'}{E \vdash_{\delta_F} \mathsf{Functor}[v : M]m : \mathsf{Funsig}(v : M)M'}$$

[M/T/STRUCT]

$$\frac{}{E \vdash_{\delta_F} \mathsf{Struct}\ \mathsf{v}_1 \triangleright v_1 : S_1 := P_1 \ldots \mathsf{v}_n \triangleright v_n : S_n := P_n\ \mathsf{End} : \mathsf{Sig}\ \mathsf{v}_1 \triangleright v_1 : S_1 \ldots \mathsf{v}_n \triangleright v_n : S_n\ \mathsf{End}}$$

Note that the difference between our definition of $\vdash_{\delta_F}$ and the definition of $\vdash_\delta$ in [32] (repeated in this thesis as Definition 1.5.1), lies in the rule [/T/ABS] which replaces the rule [/T/SPEC] allowing both kinds of specifications: $\mathsf{Ty}(M)$ and $\mathsf{Eq}(m : M)$. We disallow the latter, and hence force the reduction to happen as early as possible.

Lemma 6.3.11. *Let* $\Psi \leftrightsquigarrow \Psi'$ *be two expressions and let* q *be a shape position.*

- *If* $BV(\Psi, q) \vdash_{\delta_F} \Psi|_q : M$ *then* $BV(\Psi', q) \vdash_{\delta_F} \Psi'|_q : M'$ *where* $M \leftrightsquigarrow M'$.
- *If* $BV(\Psi, q) \vdash_{\delta_F} \Psi|_q : S$ *then* $BV(\Psi', q) \vdash_{\delta_F} \Psi'|_q : S'$ *where* $S \leftrightsquigarrow S'$.

Proof. By mutual induction on the derivation of

$$BV(\Psi, q) \vdash_{\delta_F} \Psi|_q : M \qquad\qquad \text{and} \qquad\qquad BV(\Psi, q) \vdash_{\delta_F} \Psi|_q : S$$

If the most recent rule was [/S/VAR], then $BV(\Psi, q) = E_1; v : S; E_2$ and $\Psi|_q = v$. Since $BV(\Psi, q) \leftrightsquigarrow BV(\Psi', q)$ we have $BV(\Psi', q) = E_1'; v : S'; E_2'$ with $E_1 \leftrightsquigarrow E_1'$, $S \leftrightsquigarrow S'$ and $E_2 \leftrightsquigarrow E_2'$. We also have $\Psi'|_q = v$ and hence by [/S/VAR] $E_1'; v : S'; E_2' \vdash_{\delta_F} v : S'$.

If the most recent rule was [M/T/STRUCT], we have

$$\Psi|_q = \mathsf{Struct}\ \mathtt{v}_1 \triangleright v_1 : S_1 := P_1 \ldots \mathtt{v}_n \triangleright v_n : S_n := P_n\ \mathsf{End}$$

and consequently

$$\Psi'|_q = \mathsf{Struct}\ \mathtt{v}_1 \triangleright v_1 : S_1' := P_1' \ldots \mathtt{v}_n \triangleright v_n : S_n' := P_n'\ \mathsf{End}$$

with $S_1 \leftrightsquigarrow S_1' \ldots S_n \leftrightsquigarrow S_n'$ and $P_1 \leftrightsquigarrow P_1' \ldots P_n \leftrightsquigarrow P_n'$. So using [M/T/STRUCT], we obtain $BV(\Psi', q) \vdash_{\delta_F} \Psi'|_q : \mathsf{Sig}\ \mathtt{v}_1 \triangleright v_1 : S_1' \ldots \mathtt{v}_n \triangleright v_n : S_n'\ \mathsf{End}$ and of course

$$\mathsf{Sig}\ \mathtt{v}_1 \triangleright v_1 : S_1 \ldots \mathtt{v}_n \triangleright v_n : S_n\ \mathsf{End} \leftrightsquigarrow \mathsf{Sig}\ \mathtt{v}_1 \triangleright v_1 : S_1' \ldots \mathtt{v}_n \triangleright v_n : S_n'\ \mathsf{End}$$

All other rules follow by simple use of induction hypothesis. Let us detail [/S/ACCESS] and [M/T/LAM].

If the most recent rule was [/S/ACCESS], we have $q' = q \cdot i$ and

$$BV(\Psi, q') \vdash_{\delta_F} m : \mathsf{Sig}\ \mathtt{v}_1 \triangleright v_1 : S_1 \ldots \mathtt{v}_n \triangleright v_n : S_n\ \mathsf{End}$$

where $m = \Psi|_{q'}$. By induction hypothesis,

$$BV(\Psi', q') \vdash_{\delta_F} m' : \mathsf{Sig}\ \mathtt{v}_1 \triangleright v_1 : S_1' \ldots \mathtt{v}_n \triangleright v_n : S_n'\ \mathsf{End}$$

where $m' = \Psi|_q$ and $\mathsf{Sig}\ \mathtt{v}_1 \triangleright v_1 : S_1 \ldots \mathtt{v}_n \triangleright v_n : S_n\ \mathsf{End} \leftrightsquigarrow \mathsf{Sig}\ \mathtt{v}_1 \triangleright v_1 : S_1' \ldots \mathtt{v}_n \triangleright v_n : S_n'\ \mathsf{End}$. So we have $S_i \leftrightsquigarrow S_i'$. Since $m \leftrightsquigarrow m'$ implies $m.\mathtt{v}_j \leftrightsquigarrow m'.\mathtt{v}_j$ for any j, we also have $S_i\{v_j \mapsto m.\mathtt{v}_j\}_{j=1..i-1} \leftrightsquigarrow S_i'\{v_j \mapsto m'.\mathtt{v}_j\}_{j=1..i-1}$ and using [/S/ACCESS] and the fact that $BV(\Psi', q') = BV(\Psi', q)$, we obtain the desired conclusion.

If the most recent rule was [M/T/LAM], then $q' = q \cdot 2$,

$$BV(\Psi, q') \ = \ BV(\Psi, q); \hat{v} : \mathsf{Ty}(M_1) \qquad\quad \text{and} \qquad\quad \Psi|_q = \mathsf{Functor}[\hat{v} : M_1]m$$

By induction hypothesis, $BV(\Psi', q') \vdash_{\delta_F} m' : M_2'$, where $m' = \Psi|_{q'}$, and $M_2 \leftrightsquigarrow M_2'$. Since $BV(\Psi, q') \leftrightsquigarrow BV(\Psi', q')$ we have $BV(\Psi', q') = BV(\Psi', q); \hat{v} : \mathsf{Ty}(M_1')$ with $M_1 \leftrightsquigarrow M_1'$. Hence by [M/T/LAM], $BV(\Psi', q) \vdash_{\delta_F} \mathsf{Functor}[\hat{v} : M_1']m' : \mathsf{Funsig}(\hat{v} : M_1')M_2'$ and obviously $\mathsf{Funsig}(\hat{v} : M_1)M_2 \leftrightsquigarrow \mathsf{Funsig}(\hat{v} : M_1')M_2'$ which is the expected conclusion.

Other rules are similar. ☐

Lemma 6.3.12. *If $m_1 \leftrightsquigarrow m_1'$ and $m_1 \xrightarrow{q}_{\mu\delta_F} m_2$ where q is a shape position, then there exists m_2' such that $m_1' \xrightarrow{q}_{\mu\delta_F} m_2'$ and $m_2 \leftrightsquigarrow m_2'$.*

Proof. Suppose, we have a μ redex at position q in m_1. Then $m_1|_q = \mathsf{Functor}[\hat{v} : M]\hat{m}_1\, \hat{m}_2$. Since $m_1 \leftrightsquigarrow m_1'$, $m_1'|_q = \mathsf{Functor}[\hat{v} : M']\hat{m}_1'\, \hat{m}_2'$ with $M \leftrightsquigarrow M'$, $\hat{m}_1 \leftrightsquigarrow \hat{m}_1'$ and $\hat{m}_2 \leftrightsquigarrow \hat{m}_2'$.

Therefore $(\mathsf{Functor}[\hat{v} : M']\hat{m}_1'\, \hat{m}_2') \longrightarrow_\mu \hat{m}_1'\{\hat{v} \mapsto \hat{m}_2'\}$ and the latter has similar shape to $\hat{m}_1\{\hat{v} \mapsto \hat{m}_2\}$ with is the μ reduct of $m_1|_q$.

Consequently, $m_1' \xrightarrow{q}_\mu m_2'$ with $m_2 \leftrightsquigarrow m_2'$.

Suppose, we have a δ_F-redex at position q in m_1. Then $BV(m_1, q) \vdash_{\delta_F} m_1|_q : \mathsf{Eq}(m : M)$ and by Lemma 6.3.11, $BV(m_1', q) \vdash_{\delta_F} m_1'|_q : \mathsf{Eq}(m' : M')$ with $\mathsf{Eq}(m : M) \leftrightsquigarrow \mathsf{Eq}(m' : M')$. Consequently $m \leftrightsquigarrow m'$ and therefore $m_1' \longrightarrow_{\delta_F} m_1'[m']_q$ with $m_2 = m_1[m]_q \leftrightsquigarrow m_1'[m']_q = m_2'$. ☐

Lemma 6.3.13. *Let m be a module expression without δ_F-redexes on shape positions. Then m has no δ-redexes on shape positions.*

Proof. Suppose otherwise, that there is a redex at a shape position q. We will show the following fact:

If $BV(m, q) \vdash_\delta m|_q : M$ (or $BV(m, q) \vdash_\delta m|_q : S$) then $BV(m, q) \vdash_{\delta_F} m|_q : M$ ($BV(m, q) \vdash_{\delta_F} m|_q : S$) or there is a position $\hat{q}$ such that $q \cdot \hat{q}$ is a shape position in m and $m|_{q\cdot\hat{q}}$ is a δ_F-redex.

We will prove it by induction on the derivation of the corresponding $\vdash_\delta$ judgment.

The only rule in which the conclusion does not immediately follow from induction hypothesis is [/S/SPEC] which is replaced by [/S/ABS] in the definition of $\vdash_{\delta_F}$.

From the induction hypothesis, we have either a position $\hat{q}$ such that $q \cdot \hat{q}$ is a shape position in m and $m|_{q\cdot\hat{q}}$ is a δ_F-redex or $BV(m, q) \vdash_{\delta_F} \hat{m} : S$, where $\hat{m} = m|_q$ and S is either $\mathsf{Ty}(M)$ or $\mathsf{Eq}(m' : M)$. If there is no position leading to a redex, then $S = \mathsf{Ty}(M)$. Otherwise we have a redex at position $\hat{q} = \epsilon$.

If $S = \mathsf{Ty}(M)$ then we can use [/S/ABS] instead of [/S/SPEC] to obtain $BV(m, q) \vdash_{\delta_F} \hat{m} : M$.

Other rules follow immediately from induction hypothesis. Let us take [M/T/LAM] as an example. We have $q' = q \cdot 2$, $m|_q = \mathsf{Functor}[\hat{v} : M_1]m$, $BV(m, q') = BV(m, q); \hat{v} : \mathsf{Ty}(M_1)$ and $BV(m, q); \hat{v} : \mathsf{Ty}(M_1) \vdash_\delta \hat{m} : M_2$.

From induction hypothesis, we either have a position $\hat{q}$ in $\hat{m}$ such that $q \cdot 2 \cdot \hat{q}$ is a shape position in m and $m|_{q\cdot 2\cdot\hat{q}}$ is a δ_F-redex or $BV(m, q); \hat{v} : \mathsf{Ty}(M_1) \vdash_{\delta_F} \hat{m} : M_2$.

In the first case, we have a redex at position $2 \cdot \hat{q}$ in $\mathsf{Functor}[\hat{v} : M_1]m$.

In the second case, using [/M/T/LAM], we get $BV(m, q) \vdash_{\delta_F} \mathsf{Functor}[\hat{v} : M_1]m : \mathsf{Funsig}(\hat{v} : M_1)M_2$. ☐

6.3.6 Counting positions

Before proving that every evaluation sequence is terminating, we define a subset of shape positions, called structural positions, which are very closely related to evaluation positions. By combining the results of the previous sections and confluence of the module reductions in $\mathcal{WMC}$, we show that structural positions of normal forms of translated structures do not change during evaluation. Hence the number of evaluation positions can be bounded and the termination of every evaluation sequence follows easily.

Definition 6.3.5. *Let m be a module expression. The set of* structural positions *of m, denoted by $Str(m)$ is defined as follows:*

- *ϵ is a structural position in m.*
- *If q is a structural position in m and $m|_q$ is a structure*

$$\mathsf{Struct}\ \mathtt{v}_1 \triangleright v_1 : S_1 := P_1 \ldots \mathtt{v}_n \triangleright v_n : S_n := P_n\ \mathsf{End}$$

  then $q \cdot i \cdot 2$ are structural positions for all $i = 1 \ldots n$.

If we extend $Str(\cdot)$ to terms, by setting $Str(t) = \{\epsilon\}$, we have the following equality:

$$\begin{aligned} Str(\mathsf{Struct}\ \mathtt{v}_1 \triangleright v_1 : S_1 := P_1 \ldots \mathtt{v}_n \triangleright v_n : S_n := P_n\ \mathsf{End}) &= \\ \{\epsilon\} \cup \{1 \cdot 2 \cdot q \mid q \in Str(P_1)\} \cup \{n \cdot 2 \cdot q \mid q \in Str(P_n)\} \end{aligned}$$

Of course all structural positions are shape positions and two module expressions of similar shape have identical sets of structural positions.

Lemma 6.3.14. *If $m_1 \leftrightsquigarrow m_2$ then the sets of structural positions of $\mu\delta$-normal forms of m_1 and m_2 are equal.*

Proof. Note that by Theorem 5.4.10 from [32], all $\mu\delta$-normal forms of a given module expression are β-equal. Hence their sets of structural positions are equal.

Assume we reduce m_1 using only $\mu\delta_F$-redexes on shape positions until we obtain m_1' without $\mu\delta_F$-redexes on shape positions. Using Lemma 6.3.12 we can copy these reductions in m_2, obtaining m_2' such that $m_1' \leftrightsquigarrow m_2'$ and m_2 has no $\mu\delta_F$-redexes on shape positions.

Now, if there is a $\mu\delta$-redex in m_1' by Lemma 6.3.13 the redex is at a non-shape position. Hence the term after reduction has similar shape to m_1' and has no redexes at shape positions. If we continue this reduction until a normal form m_1'', we still have $m_1' \leftrightsquigarrow m_1''$.

If we $\mu\delta$-normalize m_2', we obtain m_2'' such that $m_2' \leftrightsquigarrow m_2''$. Hence $m_1'' \leftrightsquigarrow m_2''$ and their sets of shape positions are identical.

This also means that their sets of structural positions are identical. Hence structural positions of all normal forms of m_1 and m_2 are identical. $\square$

Lemma 6.3.15. *Given a structure m, the cardinality of the set of evaluation positions of m is smaller than the cardinality of the set of its structural positions multiplied by two.*

Proof. Let us denote by $\#Ev(m)$ and $\#Str(m)$ the cardinality of the set of evaluation and structural positions of m respectively. Let us extend $\#Ev(\cdot)$ to term expressions by setting $\#Ev(t) = 0$. Let $\#Ev(m) = 0$ for m a module expression which is not a structure. Then the following equalities are true:

$$\begin{aligned} &\#Ev(\mathsf{Struct}\ \mathtt{v}_1 \triangleright v_1 : S_1 := P_1 \ldots \mathtt{v}_n \triangleright v_n : S_n := P_n\ \mathsf{End}) \\ &\qquad = \#Ev(P_1) + \cdots + \#Ev(P_n) + n + 1 \\ &\#Str(\mathsf{Struct}\ \mathtt{v}_1 \triangleright v_1 : S_1 := P_1 \ldots \mathtt{v}_n \triangleright v_n : S_n := P_n\ \mathsf{End}) \\ &\qquad = \#Str(P_1) + \cdots + \#Str(P_n) + 1 \end{aligned}$$

One can prove by induction on the structure of ψ that

$$\#Ev(\psi) \leq 2\#Str(\psi) - 1$$

If ψ is a term or a module expression which is not a structure, then the inequality holds, because $0 \leq 2 \times 1 - 1$. If ψ is a structure, then by induction hypothesis

$$\#Ev(P_1) + \cdots + \#Ev(P_n) \leq 2(\#Str(P_1) + \cdots + \#Str(P_n)) - n$$

Therefore

$$\begin{aligned}
\#Ev(&\mathsf{Struct}\ \mathtt{v}_1 \rhd v_1 : S_1 := P_1 \ldots \mathtt{v}_n \rhd v_n : S_n := P_n\ \mathsf{End}) = \\
&= \#Ev(P_1) + \cdots + \#Ev(P_n) + n + 1 \\
&\leq 2(\#Str(P_1) + \cdots + \#Str(P_n)) - n + n + 1 \\
&= 2(\#Str(P_1) + \cdots + \#Str(P_n)) + 1 \\
&= 2(\#Str(P_1) + \cdots + \#Str(P_n) + 1) - 1 \\
&= 2\#Str(\mathsf{Struct}\ \mathtt{v}_1 \rhd v_1 : S_1 := P_1 \ldots \mathtt{v}_n \rhd v_n : S_n := P_n\ \mathsf{End}) - 1
\end{aligned}$$

□

Lemma 6.3.16. *Given a correct marked structure* $\mathsf{Struct}\ Pre\ ||\ Post\ \mathsf{End}$*, we have the following inequality:* $\#Ev(\mathsf{Struct}\ Pre\ Post\ \mathsf{End}) \leq \#Ev(W(\mathsf{Struct}\ Pre\ ||\ Post\ \mathsf{End}))$*.*

Proof. Obvious. □

Lemma 6.3.17. *Let* $\vdash \mathsf{Struct}\ Pre_1\ Post_1\ \mathsf{End} : M$ *with* Pre_1 *evaluated. If*

$$Eval(Pre_1\ ||\ Post_1) \longrightarrow Eval(Pre_2\ ||\ Post_2)$$

by a **computation** *step then* $Ev(\mathsf{Struct}\ Pre_1\ Post_1\ \mathsf{End}) \subseteq Ev(\mathsf{Struct}\ Pre_2\ Post_2\ \mathsf{End})$ *and* $Ev(W(\mathsf{Struct}\ Pre_1\ ||\ Post_1\ \mathsf{End})) \subseteq Ev(W(\mathsf{Struct}\ Pre_2\ ||\ Post_2\ \mathsf{End}))$

Proof. Since some module expressions are possibly reduced to a structure, the set of evaluation positions can increase. □

Lemma 6.3.18. *Let* m *be a module expression and let* $m \longrightarrow_{\mu\delta} m'$*. Then* $Ev(m) \subseteq Ev(m')$*.*

Proof. Obvious. □

Lemma 6.3.19. *Let* $Eval(Pre_1\ ||\ Post_1) \longrightarrow \ldots \longrightarrow Eval(Pre_n\ ||\ Post_n)$ *be an evaluation sequence, where* Pre_1 *is evaluated and* $\mathsf{Struct}\ Pre_1\ Post_1\ \mathsf{End}$ *is well-typed in the empty environment.*

Then $\#Ev(m_1) \leq \cdots \leq \#Ev(m_n) < 2\#Str(\hat{m}_1)$*, where* $m_1 = \mathsf{Struct}\ Pre_1\ ||\ Post_1\ \mathsf{End}$*,* $\ldots$*,* $m_n = \mathsf{Struct}\ Pre_n\ ||\ Post_n\ \mathsf{End}$ *and* $\hat{m}_1$ *is a* $\mu\delta$*-normal form of* $W(m_1)$*.*

Proof. In this diagram, $m_i' = W(m_i)$ for $i = 1 \dots n$, m_i'' is a $\mu\delta_F$-reduced form (by reductions on shape positions) of m_i' and $\hat{m}_i$ is an $\mu\delta$-normal form of m_i'' (and m_i').

$$\begin{tikzcd}
m_1 \arrow[r, "*"] \arrow[d, dotted, "W(\cdot)"'] & m_i \arrow[r, "\mathbf{comp}"] \arrow[d, dotted] & m_{i+1} \arrow[r, "*"] \arrow[d, dotted] & m_j \arrow[r, "\mathbf{str\ end}"] \arrow[d, dotted] & m_{j+1} \arrow[r, "*"] \arrow[d, dotted] & m_n \arrow[d, dotted] \\
m_1' \arrow[dd, "\mu\delta"', "*"] & m_i' \arrow[r, "+", "\mu\delta"'] \arrow[dd, "\mu\delta"', "*"] & m_{i+1}' \arrow[dd, "\mu\delta"', "*"] & m_j' \arrow[r, phantom, "\rightsquigarrow"] \arrow[d, "\mu\delta_F"', "*"] & m_{j+1}' \arrow[d, "\mu\delta_F"', "*"] & m_n' \arrow[dd, "\mu\delta"', "*"] \\
 & & & m_j'' \arrow[r, phantom, "\rightsquigarrow"] \arrow[d, "\mu\delta"', "*"] & m_{j+1}'' \arrow[d, "\mu\delta"', "*"] & \\
\hat{m}_1 \arrow[d, dotted] & \hat{m}_i \arrow[r, leftrightarrow, "*", "\beta"'] \arrow[d, dotted] & \hat{m}_{i+1} \arrow[d, dotted] & \hat{m}_j \arrow[r, phantom, "\rightsquigarrow"] \arrow[d, dotted] & \hat{m}_{j+1} \arrow[d, dotted] & \hat{m}_n \arrow[d, dotted] \\
Str(\hat{m}_1) & Str(\hat{m}_i) & Str(\hat{m}_{i+1}) & Str(\hat{m}_j) & Str(\hat{m}_{j+1}) & Str(\hat{m}_n)
\end{tikzcd}$$

$$Str(\hat{m}_1) = \cdots = Str(\hat{m}_i) = Str(\hat{m}_{i+1}) = \cdots = Str(\hat{m}_j) = Str(\hat{m}_{j+1}) = \cdots = Str(\hat{m}_n)$$

For **simple** steps, and **structure start** steps, $m_i' = m_{i+1}'$ and therefore their normal forms are β-equal.

For **computation** steps $m_i' =_{\mu\delta} m_{i+1}'$ by Lemma 6.3.7, and hence their $\mu\delta$-normal forms are β-equal.

For **structure end**, $m_i' \rightsquigarrow m_{i+1}'$ by Lemma 6.3.10, and by Lemma 6.3.14, the sets of structural positions of their normal forms are equal.

Hence, $Str(\hat{m}_1) = \cdots = Str(\hat{m}_n)$.

The sets of evaluation positions grow to the right and downwards, and hence sizes of all sets of evaluation positions are bounded by $2\#Str(\hat{m}_n) = 2\#Str(\hat{m}_1)$, which is the desired conclusion. $\square$

Lemma 6.3.20. *Every evaluation sequence starting in a partially evaluated structure, well-typed in the empty environment is finite.*

Proof. The position of the mark in the evaluation sequence either does not move (for **computation** steps) or moves to the right (for other steps).

Since every **computation** step translates to a non-empty sequence of $\mu\delta$-reductions, the mark cannot stop at one position forever (Corollary 6.3.5).

On the other hand the number of evaluation positions in any term of the evaluation sequence is bounded by the number of structure positions of the normal form of the first term, multiplied by two (Lemma 6.3.19). Hence the mark cannot move forward forever.

Hence every reduction sequence is finite. $\square$

The following closing lemma expresses the whole idea of evaluation. Lemma 3.2.1 from Chapter 3 is its trivial consequence.

Lemma 6.3.21. *Given a structure* $m = \mathsf{Struct}\ \overline{\mathtt{v}}_1{\triangleright}\overline{v}_1 : S_1 := P_1 \ldots \overline{\mathtt{v}}_n{\triangleright}\overline{v}_n : S_n := P_n\ \mathsf{End}$ *such that* $\vdash m : M$ *for some module type* M*, there exists an evaluated structure* m' *such that* $\vdash m' : M$*. Moreover, non-module components of* m *are unchanged in* m'*.*

Proof. It is enough to evaluate $Eval(\ \|\ \overline{\mathtt{v}}_1{\triangleright}\overline{v}_1 : S_1 := P_1 \ldots \overline{\mathtt{v}}_n{\triangleright}\overline{v}_n : S_n := P_n)$ until a normal form, which is necessarily $Eval(\overline{\mathtt{v}}_1{\triangleright}\overline{v}_1 : S'_1 := P'_1 \ldots \overline{\mathtt{v}}_n{\triangleright}\overline{v}_n : S'_n := P'_n\ \|\ )$. By Lemma 6.1.2, $\overline{\mathtt{v}}_1{\triangleright}\overline{v}_1 : S'_1 := P'_1 \ldots \overline{\mathtt{v}}_n{\triangleright}\overline{v}_n : S'_n := P'_n$ is evaluated, by Lemma 6.2.3 we have $\vdash \mathsf{Struct}\ \overline{\mathtt{v}}_1{\triangleright}\overline{v}_1 : S'_1 := P'_1 \ldots \overline{\mathtt{v}}_n{\triangleright}\overline{v}_n : S'_n := P'_n\ \mathsf{End} : M$, and since none of evaluation steps changes non-module components, they remain unchanged in the result. $\square$

Chapter 7

Structure flattening

This chapter is a continuation of the results obtained in Chapter 5 about principal derivations. Its goal is to prove Lemma 3.3.1 from Chapter 3.

7.1 Environments

Lemmas 5.9.1 and 5.9.2 have a conditional form: if the environment is correct after flattening a signature or removing a functor, then any other judgment is also correct. In this section we prove that if the considered module is followed only by classic environment entries, then the environment is indeed correct. We proceed by induction on the length of the environment, inverting well-formedness judgments for specifications, switching from $\vdash$ to $\vdash_{\mathcal{P}}$ and back and using Lemmas 5.9.1 and 5.9.2. For term environment entries this is easy and for block components it sometimes requires extending the environment with more term entries. Therefore we have four lemmas: two for signature flattening and two for functor removal. The first of each pair concerns term environments and the second, all classic environments.

Lemma 7.1.1. *Let $E_1; v : \mathsf{Ty}(M); E_2; E_3 \vdash \mathtt{ok}$ where E_2 is an environment without modules, E_3 is a term environment and $M = \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$. If $E_1; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n; E_2\delta \vdash \mathtt{ok}$, where $\delta = \{\overline{v.\mathtt{v}_i} \mapsto \overline{v}_i\}$, then $E_1; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n; E_2\delta; E_3\delta \vdash \mathtt{ok}$.*

Proof. By induction on the length of E_3. If it is empty, then the conclusion is obvious. Otherwise $E_1; v : \mathsf{Ty}(M); E_2; E_3; \overline{v}' : S' \vdash \mathtt{ok}$ implies $E_1; v : \mathsf{Ty}(M); E_2; E_3 \vdash \mathtt{ok}$ and $E_1; v : \mathsf{Ty}(M); E_2; E_3 \vdash S' : \mathtt{spec}$. Let $E = E_1; v : \mathsf{Ty}(M); E_2; E_3$ and $E' = E_1; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n; E_2\delta; E_3\delta$. By induction hypothesis we have $E' \vdash \mathtt{ok}$.

Since S' is a term specification $S' = \mathsf{Ty}(t)$ or $\mathsf{Eq}(e : t)$. By inversion we have $E \vdash t : \mathrm{s}$ and $E \vdash e : t$ in case of a manifest specification. In either case we can use Lemmas 5.8.7, 5.9.1 and 5.7.1 to get $E' \vdash t\delta : \mathrm{s}$ or $E' \vdash e\delta : t\delta$ respectively. Now we can use **(Ty/Form)** or **(Eq/Form)** to obtain $E' \vdash S'\delta : \mathtt{spec}$ and **(Env/Insert)** to conclude with $E'; v' : S'\delta \vdash \mathtt{ok}$. □

Lemma 7.1.2. *Let $E_1; v : \mathsf{Ty}(M); E_2 \vdash \mathtt{ok}$ where E_2 is a classic environment and $M = \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$. Then $E_1; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n; E_2\delta \vdash \mathtt{ok}$, where $\delta = \{\overline{v.\mathtt{v}_i} \mapsto \overline{v}_i\}$.*

Proof. By induction on the length of E_2. If E_2 is empty then the result follows from the correctness of the signature M. Otherwise $E_1; v : \mathsf{Ty}(M); E_2; \overline{v}' : S' \vdash \mathtt{ok}$ implies $E_1; v : \mathsf{Ty}(M); E_2 \vdash \mathtt{ok}$ and $E_1; v : \mathsf{Ty}(M); E_2 \vdash S' : \mathtt{spec}$. Let $E = E_1; v : \mathsf{Ty}(M); E_2$ and $E' = E_1; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n; E_2\delta$. By induction hypothesis we have $E' \vdash \mathtt{ok}$. Since $E_2; \overline{v}' : S'$ is classic, S is either a term specification or an abstract block specification.

If S' is a term specification we just use Lemma 7.1.1.

In case S' is a block specification $\mathsf{Ty}(BT)$, the proof is slightly more complicated. By inversion we obtain $E \vdash BT : \mathtt{blocktype}$ and by Lemma 5.8.7 we get the validity of judgments in $Essence_P(E \vdash BT : \mathtt{blocktype})$. Let us call $\mathcal{J}$ this set of judgments. Since by Lemma 5.8.1 every judgment in $\mathcal{J}$ is of the form $E; E_3 \vdash_P J'$ where E_3 is a term environment, we can apply Lemma 7.1.1 to get $E'; E_3\delta \vdash \mathtt{ok}$ and Lemma 5.9.1 to get $E'; E_3\delta \vdash_P J\delta$

Let us call the resulting set of judgments $\mathcal{J}'$. Of course, $\mathcal{J}' = Essence_P(E' \vdash BT\delta : \mathtt{blocktype})$. Since the function $Essence_P(\cdot)$ for block correctness judgments is constructed as an inversion of the formation rule, we can use judgments in $\mathcal{J}'$ and the closure property (C3) to get $E' \vdash BT\delta : \mathtt{blocktype}$.

The closure property assures that (for the case of rewrite specification) if $\mathsf{ACC}_E(E^\Gamma, R)$ than also $\mathsf{ACC}_{E'}(E^\Gamma\delta, R\delta)$. Given $E' \vdash BT\delta : \mathtt{blocktype}$, we use **(Ty/Form)** to obtain $E' \vdash S'\delta : \mathtt{spec}$ and hence $E'; \overline{v}' : S'\delta \vdash \mathtt{ok}$ by **(Env/Insert)**. ☐

Lemma 7.1.3. *Let $E_1; v : \mathsf{Ty}(M); E_2; E_3 \vdash \mathtt{ok}$ where E_2 is an environment without modules and E_3 is a term environment and $M = \mathsf{Funsig}(\hat{v} : M_1)M_2$. If $E_1; E_2 \vdash \mathtt{ok}$ then $E_1; E_2; E_3 \vdash \mathtt{ok}$.*

Proof. By induction on the length of E_3, just like in the proof of Lemma 7.1.1. We use Lemma 5.9.2 instead of Lemma 5.9.1. ☐

Lemma 7.1.4. *Let $E_1; v : \mathsf{Ty}(M); E_2 \vdash \mathtt{ok}$ where E_2 is a classic environment and $M = \mathsf{Funsig}(\hat{v} : M_1)M_2$. Then $E_1; E_2 \vdash \mathtt{ok}$.*

Proof. The proof is very similar to the proof of Lemma 7.1.2, but the application of Lemma 5.9.1 is replaced by Lemma 5.9.2 and Lemma 7.1.1 by Lemma 7.1.3. For block specifications we also use the closure property (C4) allowing to deduce $\mathsf{ACC}_{E_1;E_2}(E^\Gamma, R)$ from $\mathsf{ACC}_{E_1;v:\mathsf{Ty}(M);E_2}(E^\Gamma, R)$ (for the case of rewrite specification). ☐

7.2 Structures

This section contains lemmas about sub-structure flattening and functor removal in structures, as well as lemmas for removing other non-classic structure components: module or block components implemented by a path or manifest. The last lemma of this section closes the conservativity proof carried out in Chapters 5, 6 and 7.

Lemma 7.2.1. *Let m be an evaluated structure, well-typed in the empty environment. If m is of the form*

$$\begin{array}{l}\mathsf{Struct}\\ \quad \overline{\mathsf{v}'}_1 \triangleright \overline{v'}_1 : S'_1 := P'_1 \ldots \overline{\mathsf{v}'}_l \triangleright \overline{v'}_l : S'_l := P'_l;\\ \quad \mathsf{v} \triangleright v : \mathsf{Ty}(M) := \mathsf{Struct}\ \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n := P_n\ \mathsf{End};\\ \quad \overline{\mathsf{v}''}_1 \triangleright \overline{v''}_1 : S''_1 := P''_1 \ldots \overline{\mathsf{v}''}_k \triangleright \overline{v''}_k : S''_k := P''_k\\ \mathsf{End}\end{array}$$

where $\overline{\mathsf{v}''}_1 \triangleright \overline{v''}_1 : S''_1 := P''_1 \ldots \overline{\mathsf{v}''}_k \triangleright \overline{v''}_k : S''_k := P''_k$ *are classic components, then the structure m' equal*

$$\begin{array}{l}\mathsf{Struct}\\ \quad \overline{\mathsf{v}'}_1 \triangleright \overline{v'}_1 : S'_1 := P'_1 \ldots \overline{\mathsf{v}'}_l \triangleright \overline{v'}_l : S'_l := P'_l;\\ \quad \overline{\tilde{\mathsf{v}}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\tilde{\mathsf{v}}}_n \triangleright \overline{v}_n : S_n := P_n;\\ \quad \overline{\mathsf{v}''}_1 \triangleright \overline{v''}_1 : S''_1\delta := P''_1\delta \ldots \overline{\mathsf{v}''}_k \triangleright \overline{v''}_k : S''_k\delta := P''_k\delta\\ \mathsf{End}\end{array}$$

where $\delta = \{\overline{v.\mathsf{v}_i} \mapsto \overline{v}\}$, *and* $\{\overline{\tilde{\mathsf{v}}}_1 \ldots \overline{\tilde{\mathsf{v}}}_n\}$ *are new labels different from* $\{\overline{\mathsf{v}'}_1 \ldots \overline{\mathsf{v}'}_l, \overline{\mathsf{v}''}_1 \ldots \overline{\mathsf{v}''}_k\}$, *is also an evaluated structure, well-typed in the empty environment.*

Proof. If m is a well-typed structure then by inversion, its primary signature is well-typed and hence the environment

$$\overline{v'}_1 : S'_1 \ldots \overline{v'}_l : S'_l; \overline{v} : \mathsf{Ty}(M); \overline{v''}_1 : S''_1 \ldots \overline{v''}_k : S''_k$$

is correct. We also know that

- $\overline{v'}_1 : S'_1 \ldots \overline{v'}_{i-1} : S'_{i-1} \vdash P'_i : S'_i$ for $i \leq m$
- $\overline{v'}_1 : S'_1 \ldots \overline{v'}_l : S'_l \vdash \mathsf{Struct}\ \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n := P_n\ \mathsf{End} : M$
- $\overline{v'}_1 : S'_1 \ldots \overline{v'}_l : S'_l; \overline{v} : \mathsf{Ty}(M); \overline{v''}_1 : S''_1 \ldots \overline{v''}_{j-1} : S''_{j-1} \vdash P''_j : S''_j$ for $j \leq k$

Since m is evaluated, we know also that $M = \mathsf{Sig}\ \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$.

In order to show that m' is well-typed, we first need to know that the environment

$$\overline{v'}_1 : S'_1 \ldots \overline{v'}_l : S'_l; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n; \overline{v''}_1 : S''_1\delta \ldots \overline{v''}_k : S''_k\delta$$

is correct. This follows directly from Lemma 7.1.2. Consequently, the primary signature of m' is correct. We are left to prove that the implementations of components of m' verify their respective specifications. From the first point above we know that

- $\overline{v'}_1 : S'_1 \ldots \overline{v'}_{i-1} : S'_{i-1} \vdash P'_i : S'_i$ for $i \leq m$

From the second point we know that components of

$$\mathsf{Struct}\ \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n := P_n\ \mathsf{End}$$

are correct. This gives us:

- $\overline{v'}_1 : S'_1 \ldots \overline{v'}_l : S'_l; \overline{v}_1 : S_1 \ldots \overline{v}_{i-1} : S_{i-1} \vdash P_i : S_i$ for $i \leq n$

We are left with the last m components. We have to show that

- $\overline{v'}_1 : S'_1 \ldots \overline{v'}_l : S'_l; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n; \overline{v''}_1 : S''_1\delta \ldots \overline{v''}_{j-1} : S''_{j-1}\delta \vdash P''_j\delta : S''_j\delta$ for $j \leq k$

Let us pick an arbitrary j. Since S''_j is classic, it is either a term specification or a block specification. If it is a term specification, then by Lemmas 5.8.7, 5.9.1 and 5.7.1, we get the desired conclusion.

If S'' is a block specification then it is of the form $\mathsf{Ty}(BT)$ We already know that $S''_j\delta$ is a correct specification and hence

$$\overline{v'}_1 : S'_1 \ldots \overline{v'}_l : S'_l; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n; \overline{v''}_1 : S''_1\delta \ldots \overline{v''}_{j-1} : S''_{j-1}\delta \vdash BT\delta : \mathtt{blocktype}$$

If BT is an inductive specification $\mathsf{IndSpec}(E^I := E^C)$, then P''_j is necessarily $\mathsf{Ind}(E^I := E^C)$. Since $\mathsf{IndSpec}(E^I := E^C)\delta$ is correct, then we can use the rule **(IndSpec/Ind)** to get the desired conclusion.

If BT is a rewrite specification $\mathsf{RewSpec}(E^\Gamma, R)$, then P''_j is necessarily $\mathsf{Rew}(E^\Gamma, R)$. Like previously, we can apply the rule **(RewSpec/Rew)**, because the side-condition is satisfied. Indeed, by the closure property (C3), we can deduce $\mathsf{COMP}_{E'}(E^\Gamma\delta, R\delta)$ from $\mathsf{COMP}_E(E^\Gamma, R)$, where E and E' are

$$\overline{v'}_1 : S'_1 \ldots \overline{v'}_l : S'_l; v : \mathsf{Ty}(M); \overline{v''}_1 : S''_1 \ldots \overline{v''}_{j-1} : S''_{j-1}$$

and

$$\overline{v'}_1 : S'_1 \ldots \overline{v'}_l : S'_l; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n; \overline{v''}_1 : S''_1\delta \ldots \overline{v''}_{j-1} : S''_{j-1}\delta$$

respectively.

Since j was arbitrary we now have all premises of the **(Sig/Struct)** rule allowing to derive the well-typedness of the structure m' in the empty environment. □

Lemma 7.2.2. *Let m be an evaluated structure, well-typed in the empty environment. If m is of the form*

$$\begin{array}{l}\mathsf{Struct}\\ \quad \overline{\mathsf{v}'}_1 \triangleright \overline{v'}_1 : S'_1 := P'_1 \ldots \overline{\mathsf{v}'}_l \triangleright \overline{v'}_l : S'_l := P'_l;\\ \quad \mathsf{v} \triangleright v : \mathsf{Ty}(\mathsf{Funsig}(\hat{v} : M_1)M_2) := m;\\ \quad \overline{\mathsf{v}''}_1 \triangleright \overline{v''}_1 : S''_1 := P''_1 \ldots \overline{\mathsf{v}''}_k \triangleright \overline{v''}_k : S''_k := P''_k\\ \mathsf{End}\end{array}$$

where $\overline{\mathsf{v}''}_1 \triangleright \overline{v''}_1 : S''_1 := P''_1 \ldots \overline{\mathsf{v}''}_k \triangleright \overline{v''}_k : S''_k := P''_k$ *are classic components, then the structure* m' *equal*

$$\begin{array}{l}\mathsf{Struct}\\ \quad \overline{\mathsf{v}'}_1 \triangleright \overline{v'}_1 : S'_1 := P'_1 \ldots \overline{\mathsf{v}'}_l \triangleright \overline{v'}_l : S'_l := P'_l;\\ \quad \overline{\mathsf{v}''}_1 \triangleright \overline{v''}_1 : S''_1 := P''_1 \ldots \overline{\mathsf{v}''}_k \triangleright \overline{v''}_k : S''_k := P''_k\\ \mathsf{End}\end{array}$$

is also an evaluated structure, well-typed in the empty environment.

Proof. The proof of this lemma is very similar to the proof of Lemma 7.2.1. We use Lemma 7.1.4 to deduce that the primary signature of m' is well-typed. To get the fact that all implementations satisfy their respective specifications, for term components we use Lemmas 5.8.7, 5.9.2 and 5.7.1 and for block components, inductive definitions are handled trivially and correctness of definitions by rewriting follows from the closure property (C4), allowing to deduce $\mathsf{COMP}_{E'}(E^{\Gamma}, R)$ from $\mathsf{COMP}_{E}(E^{\Gamma}, R)$, where E and E' are

$$\overline{v'}_1 : S'_1 \ldots \overline{v'}_l : S'_l; v : \mathsf{Ty}(\mathsf{Funsig}(\hat{v} : M_1)M_2); \overline{v''}_1 : S''_1 \ldots \overline{v''}_{j-1} : S''_{j-1}$$

and

$$\overline{v'}_1 : S'_1 \ldots \overline{v'}_l : S'_l; \overline{v''}_1 : S''_1 \ldots \overline{v''}_{j-1} : S''_{j-1}$$

respectively. □

Lemma 7.2.3. *Let* m *be an evaluated structure, well-typed in the empty environment. If* m *is of the form*

$$\begin{array}{l}\mathsf{Struct}\\ \quad \overline{\mathsf{v}'}_1 \triangleright \overline{v'}_1 : S'_1 := P'_1 \ldots \overline{\mathsf{v}'}_l \triangleright \overline{v'}_l : S'_l := P'_l;\\ \quad \overline{\mathsf{v}} \triangleright \overline{v} : \mathsf{Eq}(\overline{p} : \Psi) := \overline{\phi};\\ \quad \overline{\mathsf{v}''}_1 \triangleright \overline{v''}_1 : S''_1 := P''_1 \ldots \overline{\mathsf{v}''}_k \triangleright \overline{v''}_k : S''_k := P''_k\\ \mathsf{End}\end{array}$$

where $\overline{\mathsf{v}''}_1 \triangleright \overline{v''}_1 : S''_1 := P''_1 \ldots \overline{\mathsf{v}''}_k \triangleright \overline{v''}_k : S''_k := P''_k$ *are classic components, and* $\overline{\mathsf{v}} \triangleright \overline{v} : \mathsf{Eq}(\overline{p} : \Psi) := \overline{\phi}$ *is either a block component or a module component then the structure* m' *equal*

$$\begin{array}{l}\mathsf{Struct}\\ \quad \overline{\mathsf{v}'}_1 \triangleright \overline{v'}_1 : S'_1 := P'_1 \ldots \overline{\mathsf{v}'}_l \triangleright \overline{v'}_l : S'_l := P'_l;\\ \quad \overline{\mathsf{v}''_1} \triangleright \overline{v''_1} : S''_1\delta := P''_1\delta \ldots \overline{\mathsf{v}''_k} \triangleright \overline{v''_k} : S''_k\delta := P''_k\delta\\ \mathsf{End}\end{array}$$

where $\delta = \{\overline{v} \mapsto \overline{p}\}$ *is also an evaluated structure, well-typed in the empty environment.*

Proof. This is an easy consequence of Lemma 4.1.5. □

Lemma 7.2.4. *Let m be an evaluated structure, well-typed in the empty environment. If m is of the form*

$$\begin{array}{l}\mathsf{Struct}\\ \quad \overline{\mathsf{v}'}_1 \triangleright \overline{v'}_1 : S'_1 := P'_1 \ldots \overline{\mathsf{v}'}_l \triangleright \overline{v'}_l : S'_l := P'_l;\\ \quad \overline{\mathsf{v}} \triangleright \overline{v} : \mathsf{Ty}(\Psi) := \overline{p};\\ \quad \overline{\mathsf{v}''}_1 \triangleright \overline{v''}_1 : S''_1 := P''_1 \ldots \overline{\mathsf{v}''}_k \triangleright \overline{v''}_k : S''_k := P''_k\\ \mathsf{End}\end{array}$$

where $\overline{\mathsf{v}''}_1 \triangleright \overline{v''}_1 : S''_1 := P''_1 \ldots \overline{\mathsf{v}''}_k \triangleright \overline{v''}_k : S''_k := P''_k$ are classic components, and $\overline{\mathsf{v}} \triangleright \overline{v} : \mathsf{Ty}(\Psi) := \overline{p}$ is either a block component or a module component, then the structure m' equal

$$\begin{array}{l}\mathsf{Struct}\\ \quad \overline{\mathsf{v}'}_1 \triangleright \overline{v'}_1 : S'_1 := P'_1 \ldots \overline{\mathsf{v}'}_l \triangleright \overline{v'}_l : S'_l := P'_l;\\ \quad \overline{\mathsf{v}''_1} \triangleright \overline{v''_1} : S''_1\delta := P''_1\delta \ldots \overline{\mathsf{v}''_k} \triangleright \overline{v''_k} : S''_k\delta := P''_k\delta\\ \mathsf{End}\end{array}$$

where $\delta = \{\overline{v} \mapsto \overline{p}\}$ is also an evaluated structure, well-typed in the empty environment.

Proof. This is again an easy consequence of Lemma 4.1.5. □

The following lemma is a formal counter-part of Lemma 3.3.1, presented in Chapter 3.

Lemma 7.2.5. *Let m be an evaluated structure, well-typed in the empty environment. If m is of the form*

$$\begin{array}{l}\mathsf{Struct}\\ \quad \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n := P_n;\\ \quad \overline{\mathsf{v}'}_1 \triangleright \overline{v'}_1 : S'_1 := P'_1 \ldots \overline{\mathsf{v}'}_k \triangleright \overline{v'}_k : S'_k := P'_k\\ \quad \mathsf{v} \triangleright v : \mathsf{Ty}(t) := e;\\ \mathsf{End}\end{array}$$

where $\overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n := P_n$ are classic components and t is typable is the environment $\overline{v}_1 : S_1 \ldots \overline{v}_n : S_n$, then there exists a classic structure m', well-typed in the empty environment, and of the following form:

$$\begin{array}{l}\mathsf{Struct}\\ \quad \overline{\mathsf{v}}_1 \triangleright \overline{v}_1 : S_1 := P_1 \ldots \overline{\mathsf{v}}_n \triangleright \overline{v}_n : S_n := P_n;\\ \quad \overline{\mathsf{v}''}_1 \triangleright \overline{v''}_1 : S''_1 := P''_1 \ldots \overline{\mathsf{v}''}_l \triangleright \overline{v''}_l : S''_l := P''_l\\ \quad \mathsf{v} \triangleright v : \mathsf{Ty}(t) := e';\\ \mathsf{End}\end{array}$$

Proof. We proceed from right to left to eliminate all non-classic components of m.

Let us consider the last non-classic component. It is either a block component or a module component. If it contains a manifest specification, then we use Lemma 7.2.3. If it is an abstract module or block component, then if its implementation is a path or a sequence of paths, we use Lemma 7.2.4. If it is an abstract module component then its implementation can also be a functor or an evaluated structure. In the first case we use Lemma 7.2.2, and in the second case, Lemma 7.2.1.

If it is an abstract block component then it can be implemented by a basic block which does not match exactly its specification, for example $\overline{\mathsf{v}}\triangleright\overline{v} : \mathsf{Ty}(\mathsf{IndSpec}(E^{I'} := E^{C'})) := \mathsf{Ind}(E^I := E^C)$. In this case the primary block type of the basic block is correct and convertible with the block type in the specification. Hence by Lemma 4.1.7 we can simply change the specification obtaining the classic component. In our example, we would get $\overline{\mathsf{v}}\triangleright\overline{v} : \mathsf{Ty}(\mathsf{IndSpec}(E^I := E^C)) := \mathsf{Ind}(E^I := E^C)$.

In all cases the total number of components and sub-components of m decreases and therefore the procedure is necessarily finite. Note also, that substitutions appearing in lemmas used in the proof do not change the type of the last component t, as the domains of the substitutions contain names of non-classic components of m, which all appear in m after the index n. Since t is typable in the environment $\overline{v}_1 : S_1 \ldots \overline{v}_n : S_n$ it cannot contain names of non-classic components and therefore none of the substitutions can change it. $\square$

Chapter 8

Decidability

In this chapter we prove that decidability of the calculus with inductive types and definitions by rewriting implies decidability of the calculus with named modules.

The most important part consists in showing that type-checking terms in an arbitrary environment can be reduced in polynomial time into type-checking terms in a classic environment, i.e. without modules and manifest blocks. We proceed in two steps: first we show how to remove modules, and then manifest blocks.

As far as the first step is considered we reuse signature flattening and functor removal operations defined in the previous chapters. It is easy to show that these operations are bidirectional, i.e. a term judgment holds after the operation if and only if it holds before the operation. Unfortunately this is not the case for the treatment of manifest modules, presented in Chapter 7. Therefore we introduce a more complex way of eliminating manifest modules, which is bidirectional.

This chapter is organised as follows. Section 8.1 shows that a sequence of paths satisfies an inductive specification in $\vdash$ only if it does in $\vdash_{\mathcal{P}}$. This result is later used to show decidability of satisfaction of inductive specifications and to show that all modules can be removed from the environment before the manifest blocks. Note that in Chapter 7 removing of both these kinds of components is done simultaneously.

Section 8.2 presents the bidirectional way of eliminating manifest modules. Section 8.3 and 8.4 contain all lemmas needed to show the correctness of respectively the first and the second phase of the reduction: removing of modules and removing of manifest blocks. Section 8.5 completes these results with the brief description of the module typing algorithm given in the proof of Theorem 2.

8.1 Principal derivations of inductive types

This section shows that a sequence of paths can satisfy an inductive specification only if it is declared as an inductive block, i.e. the satisfaction of an inductive specification is derivable in $\vdash_{\mathcal{P}}$.

The proof of this property follows very closely the reasoning presented in Section 5.8, starting by the definition below which is very similar to Definition 5.8.2, but much simpler.

Definition 8.1.1. *The* inductive essence *of a judgment $E \vdash J$, written $\mathit{IndEss}(E \vdash J)$ is defined as follows:*

$\mathit{IndEss}(E \vdash \overline{p} : \mathsf{IndSpec}(E^I := E^C)) =$
$\quad \{E \vdash_P \overline{p} : \mathsf{IndSpec}(E^{I'} := E^{C'})\} \cup \{E \vdash \mathsf{IndSpec}(E^I := E^C) \approx \mathsf{IndSpec}(E^{I'} := E^{C'})\}$
$\quad$ *for some* $E^{I'}$ *and* $E^{C'}$

$\mathit{IndEss}(E \vdash p : \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}) =$
$\quad \bigcup\{\mathit{IndEss}(E \vdash \overline{p.\mathtt{v}_i} : S_i\{\overline{v}_j \mapsto \overline{p.\mathtt{v}_j}\})\ \ \text{for } i = 1 \ldots n\}$

$\mathit{IndEss}(E \vdash \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End} <: \mathsf{Sig}\ \overline{\mathtt{v}'}_1 \triangleright \overline{v'}_1 : S'_1 \ldots \overline{\mathtt{v}'}_m \triangleright \overline{v'}_m : S'_m\ \mathsf{End}) =$
$\quad \bigcup\{\mathit{IndEss}(E; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n \vdash \overline{v}'_i : S'_i)\ \ \text{for } i = 1 \ldots m\}$
$\quad$ *where* $\{\overline{\mathtt{v}}_1 \ldots \overline{\mathtt{v}}_n\} \supseteq \{\overline{\mathtt{v}}'_1 \ldots \overline{\mathtt{v}}'_m\}$ *and* $\mathtt{v}_i^j = \mathtt{v'}_{i'}^{j'}$ *implies* $v_i^j = v'^{j'}_{i'}$

$\mathit{IndEss}(E \vdash \overline{p} : \mathsf{Ty}(\Psi)) =$
$\quad \mathit{IndEss}(E \vdash \overline{p} : \Psi)$

$\mathit{IndEss}(E \vdash \overline{p} : \mathsf{Eq}(\overline{\phi} : \Psi)) =$
$\quad \mathit{IndEss}(E \vdash \overline{p} : \Psi)$

$\mathit{IndEss}(E \vdash J) = \emptyset$ *otherwise*

The lemmas below correspond to Lemmas 5.8.1, 5.8.2, 5.8.4, 5.8.5 and 5.8.7 respectively.

Lemma 8.1.1. *For any judgment $E \vdash J$, we have*

$$\begin{aligned}\mathit{IndEss}(E \vdash J) = \mathit{IndEss}(E; E'_1 \vdash \overline{p}_1 : \mathsf{IndSpec}(E_1^I := E_1^C)) \cup \ldots \\ \cdots \cup \mathit{IndEss}(E; E'_n \vdash \overline{p}_n : \mathsf{IndSpec}(E_n^I := E_n^C))\end{aligned}$$

for some $\overline{p}_1 \ldots \overline{p}_n$, $E_1^I \ldots E_n^I$ and $E_1^C \ldots E_n^C$.

Moreover this equality is stable under substitution, i.e. if $E = E_1; v : S; E_2$ and given a substitution $\delta = \{v \mapsto p\}$, we have

$$\begin{aligned}\mathit{IndEss}(E_1; E_2\delta \vdash J\delta) = \mathit{IndEss}(E_1; E_2\delta; E'_1\delta \vdash \overline{p}_1\delta : \mathsf{IndSpec}(E_1^I\delta := E_1^C\delta)) \cup \ldots \\ \cdots \cup \mathit{IndEss}(E_1; E_2\delta; E'_n\delta \vdash \overline{p}_n\delta : \mathsf{IndSpec}(E_n^I\delta := E_n^C\delta))\end{aligned}$$

Proof. By inspection of the definition of $\mathit{IndEss}(\cdot)$. □

Lemma 8.1.2. *If $E \vdash_P J$ then all judgments in $\mathit{IndEss}(E \vdash J)$ are true.*

Proof. For judgments of the form $E \vdash_P \overline{p} : \mathsf{IndSpec}(E^I := E^C)$, we get the conclusion by reflexivity of $\approx$ (Lemma 4.1.12).

For other judgments of the form $\psi : \Phi$, we proceed by induction on the weight of Φ (see Definition 5.8.1)

For all other judgments in $\vdash_P$ the conclusion is trivial. □

Lemma 8.1.3. *If* $E_1; \overline{v} : S; E_2 \vdash_P J$ *and* $E_1 \vdash \overline{p} : S$ *and all judgments in* $\mathit{IndEss}(E_1 \vdash \overline{p} : S)$ *hold, then all judgments in* $\mathit{IndEss}(E_1; E_2\delta \vdash J\delta)$ *hold as well, where* $\delta = \{\overline{v} \mapsto \overline{p}\}$.

Proof. The proof is almost identical to the proof of Lemma 5.8.4. We proceed by induction on the derivation of $E_1; \overline{v} : S; E_2 \vdash_P J$.

If the last rule of the derivation was **(P/Env/Lookup)** and the path that was looked up was $\overline{v}$ then we use the assumption. Otherwise we can apply the rule **(P/Env/Lookup)** to the correct environment $E_1; E_2\delta$ and get $E_1; E_2\delta \vdash_P \overline{v}' : S'\delta$ and all judgments in $\mathit{IndEss}(E_1; E_2\delta \vdash_P \overline{v}' : S'\delta)$ by Lemma 8.1.2.

The other interesting case is the rule **(P/Sig/Access)**, which is handled exactly as in Lemma 5.8.4.

The rest of the rules can be handled by induction hypothesis. □

Lemma 8.1.4. *Let* $E_1; \overline{v} : S; E_2 \vdash J$ *and let all judgments in* $\mathit{IndEss}(E_1; \overline{v} : S; E_2 \vdash J)$ *hold. If* $E_1 \vdash \overline{p} : S$ *and all judgments in* $\mathit{IndEss}(E_1 \vdash \overline{p} : S)$ *hold, then all judgments in* $\mathit{IndEss}(E_1; E_2\delta \vdash J\delta)$ *also hold for* $\delta = \{\overline{v} \mapsto \overline{p}\}$.

Proof. Given Lemma 8.1.1, it is enough to show the above for judgments of the form $E_1; \overline{v} : S; E_2; E' \vdash \overline{p} : \mathsf{IndSpec}(E^I := E^C)$. By Definition 8.1.1,

$$\begin{aligned}\mathit{IndEss}(E_1; \overline{v} : S; E_2; E' \vdash \overline{p} : \mathsf{IndSpec}(E^I := E^C)) \ = \\ \{E_1; \overline{v} : S; E_2; E' \vdash_P \overline{p} : \mathsf{IndSpec}(E^{I'} := E^{C'})\} \ \cup \\ \{E_1; \overline{v} : S; E_2; E' \vdash \mathsf{IndSpec}(E^I := E^C) \approx \mathsf{IndSpec}(E^{I'} := E^{C'})\}\end{aligned}$$

for some $E^{I'}$ and $E^{C'}$. By Lemma 8.1.3, we have the validity of judgments in

$$\mathit{IndEss}(E_1; E_2\delta; E'\delta \vdash \overline{p}\delta : \mathsf{IndSpec}(E^{I'}\delta := E^{C'}\delta))$$

i.e.

$$\begin{aligned}&E_1; E_2\delta; E'\delta \vdash_P \overline{p}\delta : \mathsf{IndSpec}(E^{I''} := E^{C''}) \qquad \text{and} \\ &E_1; E_2\delta; E'\delta \vdash \mathsf{IndSpec}(E^{I'}\delta := E^{C'}\delta) \approx \mathsf{IndSpec}(E^{I''} := E^{C''})\end{aligned} \tag{8.1}$$

for some $E^{I''}$ and $E^{C''}$. By Lemma 4.1.5 we also have

$$E_1; E_2\delta; E'\delta \vdash \mathsf{IndSpec}(E^I\delta := E^C\delta) \approx \mathsf{IndSpec}(E^{I'}\delta := E^{C'}\delta)$$

which by transitivity of conversion (Lemma 4.1.12) gives

$$E_1; E_2\delta; E'\delta \vdash \mathsf{IndSpec}(E^I\delta := E^C\delta) \approx \mathsf{IndSpec}(E^{I''} := E^{C''}) \tag{8.2}$$

Putting (8.1) and (8.2) together, we have proved the validity of the judgments in

$$\mathit{IndEss}(E_1; E_2\delta; E'\delta \vdash \overline{p}\delta : \mathsf{IndSpec}(E^I\delta := E^C\delta))$$

which concludes the proof. □

Lemma 8.1.5. *If $E \vdash J$ then all judgments in IndEss($E \vdash J$) hold. In particular, if $E \vdash \overline{p} : \mathsf{IndSpec}(E^I := E^C)$ then*

$$E \vdash_P \overline{p} : \mathsf{IndSpec}(E^{I'} := E^{C'}) \quad \textit{and} \quad E \vdash \mathsf{IndSpec}(E^I := E^C) \approx \mathsf{IndSpec}(E^{I'} := E^{C'})$$

for some $E^{I'}$ and $E^{C'}$.

Proof. The proof is done by induction on the derivation of $E \vdash J$ and is very similar to the proof of Lemma 5.8.7.

If the last rule used to derive $E \vdash J$ was **(Env/Lookup)**, then the judgment $E \vdash_P J$ can be derived using **(P/Env/Lookup)** and we can use Lemma 8.1.2.

For term rules we have nothing to prove.

For module rules, the most interesting rule is **(Subsumption)** for the case when m is a path and M_1 and M_2 are signatures. We proceed exactly like in Lemma 5.8.7, using Lemma 8.1.4.

The rules **(Sig/Access)**, **(Sig/Sub)** and **(Strengthening)** are handled immediately by induction hypothesis and for other module rules, there is nothing to prove.

Specification rules **(Ty/Sat)**, **(Ty/Type)** and **(Eq/Sat)**, **(Eq/Type)** are handled by induction hypothesis and for others there is nothing to prove.

For block rules, the only interesting one is **(Block/Conv)**, for the case when B is a sequence of paths $\overline{p}$ and BT and BT' are both inductive specifications:

$$\frac{\begin{array}{c} E \vdash \overline{p} : \mathsf{IndSpec}(E^I := E^C) \\ E \vdash \mathsf{IndSpec}(E^{I'} := E^{C'}) : \texttt{blocktype} \\ E \vdash \mathsf{IndSpec}(E^I := E^C) \approx \mathsf{IndSpec}(E^{I'} := E^{C'}) \end{array}}{E \vdash \overline{p} : \mathsf{IndSpec}(E^{I'} := E^{C'})}$$

By induction hypothesis, we get

$$E \vdash_P \overline{p} : \mathsf{IndSpec}(E^{I''} := E^{C''}) \quad \text{and} \quad E \vdash \mathsf{IndSpec}(E^I := E^C) \approx \mathsf{IndSpec}(E^{I''} := E^{C''})$$

for some $E^{I''}$ and $E^{C''}$. From the second judgment and the third premise, symmetry and transitivity (Lemma 4.1.12), we get

$$E \vdash \mathsf{IndSpec}(E^{I'} := E^{C'}) \approx \mathsf{IndSpec}(E^{I''} := E^{C''})$$

which concludes the proof since there is nothing to prove for convertibility rules. □

8.2 Elimination of manifest modules

We show that it is possible to transform a manifest module component $v : \mathsf{Eq}(p : M)$ into an abstract module component $v : \mathsf{Ty}(M/p)$, provided that it is the last module component in the environment. The lemmas in this section correspond to Lemmas 5.9.1, 7.1.1 and 7.1.2 for signature flattening and to Lemmas 5.9.2, 7.1.3 and 7.1.4 for functor removal.

The first lemma is a bit more complex than its counterparts (Lemmas 5.9.1 and 5.9.2), because we have to take care of equalities resulting from $v \approx p$, derivable in the initial environment, but not in the environment after the elimination operation. That is why we need the following auxiliary definition.

Definition 8.2.1. *Given a correct environment E and a module path p, The* set of labels *of the path p in the environment E, denoted by $Lab_E(p)$, is defined as follows: if $E \vdash_P p : M_P$ then*

- *if* $M_P = \mathsf{Funsig}(\hat{v} : M_1)M_2$ *then* $Lab_E(p) = \emptyset$
- *if* $M_P = \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$ *then* $Lab_E(p) = \{\overline{\mathtt{v}}_1 \ldots \overline{\mathtt{v}}_n\}$

Lemma 8.2.1. *Let $E = E_1; v : \mathsf{Eq}(p : M); E_2$ and $E' = E_1; v : \mathsf{Ty}(M/p); E_2$ where E_2 is an environment without modules. If $E \vdash_P J$ and $E' \vdash \mathtt{ok}$ then J does not contain subexpressions $\mathsf{Eq}(v : \hat{M})$ and*

- $E' \vdash p \approx p'$ *and* $E' \vdash v.\mathtt{v}_i^j \approx p'.\mathtt{v}_i^j$ *for all* $\mathtt{v}_i^j \in Lab_E(v) \cap Lab_E(p')$
  if J is of the form $v \approx p'$ or $p' \approx v$ for $p' \neq v$
- *nothing*
  if $J = v : \mathsf{Eq}(p : M)$ *or* $v \approx v$
- $E' \vdash J$
  otherwise

Proof. We start by proving the following two facts:

- $Lab_E(v) \subseteq Lab_E(p)$
- for all $\mathtt{v}_i^j \in Lab_E(v)$ we have $E' \vdash v.\mathtt{v}_i^j \approx p.\mathtt{v}_i^j$

The first statement follows by the analysis of the module types of p. By inversion we get $E \vdash p : M$ and therefore by Lemma 5.6.5, $E \vdash M_P/p <: M$, where $E \vdash_P p : M_P$. So either both Mp and M are functor types and $Lab_E(v) = Lab_E(p) = \emptyset$ or they are both signatures. It this case obviously M_P has more labels than M.

The second statement is trivial if M is a functor type. Otherwise

$$\begin{aligned} M &= \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End} \\ &\text{and} \\ M/p &= \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1/\overline{p.\mathtt{v}_1} \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n/\overline{p.\mathtt{v}_n}\ \mathsf{End} \end{aligned}$$

and Since $E_1; v : \mathsf{Ty}(M/p) \vdash p : M$ and $E_1; v : \mathsf{Ty}(M/p) \vdash M : \mathtt{modtype}$, by Lemma 4.1.14 we also have

$$E_1; v : \mathsf{Ty}(M/p); \overline{v}_1 : S_1/\overline{p.\mathtt{v}_1} \ldots \overline{v}_i : S_i/\overline{p.\mathtt{v}_i} \vdash v_i^j \approx p.\mathtt{v}_i^j$$

for all $i = 1 \dots n$ and any j. Since $E_1; v : \mathsf{Ty}(M/p) \vdash v.\mathtt{v}_i : S_i/\overline{p.\mathtt{v}_i}\{v_j \mapsto \overline{v.\mathtt{v}_j}\}$, by Lemma 4.1.6 we get

$$E_1; v : \mathsf{Ty}(M/p) \vdash v.\mathtt{v}_i^j \approx p.\mathtt{v}_i^j$$

Now we can do the proof by induction on the derivation of $E \vdash_P J$.

If the last rule in the derivation is **(P/Env/Lookup)**, there are two cases to consider. Either the sequence of variables which is looked up is different from v or the variable which is looked up is v. In the first case we can do the same in E' and the absence of subexpressions $\mathsf{Eq}(v : \hat{M})$ follows from the v does not appear in E_1 and E_2 has no module components. In the second case, we do not have to prove anything.

If the last rule is **(P/Sig/Access)** or **(P/Ty/Type)**, then it simply follows by induction hypothesis.

To analyze the rule **(P/Eq/Type)** we have to consider two cases. If the premise is different from $v : \mathsf{Eq}(p : M)$ we can proceed by induction hypothesis. If the premise is $v : \mathsf{Eq}(p : M)$, then we have to show $E' \vdash v : M$. But this holds, since $E_1 \vdash p : M$ implies $E_1 \vdash M/p <: M$ by Lemma 4.1.14 and since E' is an extension of E_1, we can apply **(Subsumption)** in order to get $E' \vdash v : M$ from $E' \vdash v : M/p$.

If the last rule was **(P/Eq/Comp)** then either the premise is different from $v : \mathsf{Eq}(p : M)$ and we can proceed by induction hypothesis or it is $v : \mathsf{Eq}(p : M)$. In the latter case we already have $E' \vdash v.\mathtt{v}_i^j \approx p.\mathtt{v}_i^j$ for all $\mathtt{v}_i^j \in Lab_E(v)$ and since $Lab_E(v) \subseteq Lab_E(p)$, we have $Lab_E(v) \cap Lab_E(p) = Lab_E(v)$.

If the last rule was a term rule, then we can simply apply induction hypothesis, using Lemmas 5.7.3 and 5.7.4 to establish the correctness of bigger environments in rules **(P/Term/Prod)** and **(P/Term/Abs)**. In the rule **(P/Term/Conv)** we cannot have $E \vdash v \approx p'$ as a premise since v is a module path.

If block rules were used, then we simply apply induction hypothesis.

Conversion rules for paths have to be carefully studied, particularly the transitivity rule:

$$\frac{E \vdash p' \approx p'' \quad E \vdash p'' \approx p'''}{E \vdash p' \approx p'''}$$

We will consider four cases depending on which of p', p'', p''' are equal v.

- If none of p', p'', p''' is equal v then we proceed by induction hypothesis.

- If two or three of p', p'', p''' are equal v then either we do not have to prove anything or the conclusion is one of the premises and we proceed by induction hypothesis.

- If one of p', p'', p''' is equal v then the cases $p' = v$ and $p''' = v$ are symmetric. We present the second one. By induction hypothesis, we have $E' \vdash p' \approx p''$ and $E' \vdash p'' \approx p$ so using transitivity, we get $E' \vdash p' \approx p$.

  We also have to prove $E' \vdash v.\mathtt{v}_i^j \approx p'.\mathtt{v}_i^j$ for all $\mathtt{v}_\mathtt{i}^\mathtt{j} \in Lab_E(v) \cap Lab_E(p')$. We will use p again.

  Note that for all $\mathtt{v}_\mathtt{i}^\mathtt{j} \in Lab_E(p) \cap Lab_E(p')$ both $p.\mathtt{v}_i^j$ and $p'.\mathtt{v}_i^j$ are well-typed.

Hence we can use the rule **(Conv/Path)** to show $E' \vdash p.\mathtt{v}_i^j \approx p'.\mathtt{v}_i^j$ for all $\mathtt{v}_\mathtt{i}^\mathtt{j} \in Lab_E(p) \cap Lab_E(p')$ using $E' \vdash p \approx p'$, which we have already proved.

Now, using the facts proved in the beginning of the proof and transitivity, we can get $E' \vdash v.\mathtt{v}_i^i \approx p'.\mathtt{v}_i^j$ for all $\mathtt{v}_\mathtt{i}^\mathtt{j} \in Lab_E(v) \cap Lab_E(p')$.

- If $p'' = v$, then by induction hypothesis $E' \vdash p' \approx p$ and $E' \vdash p \approx p'''$ so using transitivity, we get $E' \vdash p' \approx p'''$.

The next interesting rule is **(P/Conv/Path)** if exactly one of concerned paths is v. We have

$$\frac{E \vdash_P v.\mathtt{v}_i^j : \Psi \quad E \vdash_P v \approx p' \quad E \vdash_P p'.\mathtt{v}_i^j : \Psi'}{E \vdash_P v.\mathtt{v}_i^j \approx p'.\mathtt{v}_i^j}$$

Since both $v.\mathtt{v}_i^j$ and $p'.\mathtt{v}_i^j$ are typable we have $\mathtt{v}_i^j \in Lab_E(v) \cap Lab_E(p')$ and by induction hypothesis about the second judgment, we get the desired conclusion.

If none of the premises is v, then we simply proceed by induction hypothesis. Other convertibility rules are also handled by induction hypothesis since none of the convertibility premises contains v as they are all term judgments. □

Lemma 8.2.2. *Let $E_1; v : \mathsf{Eq}(p : M); E_2; E_3 \vdash \mathtt{ok}$ where E_2 is an environment without modules and E_3 is a term environment. If $E_1; v : \mathsf{Ty}(M/p); E_2 \vdash \mathtt{ok}$ then $E_1; v : \mathsf{Ty}(M/p); E_2; E_3 \vdash \mathtt{ok}$.*

Proof. By induction on the length of E_3. If it is empty, then the conclusion is obvious.

Otherwise $E_1; v : \mathsf{Eq}(p : M); E_2; E_3; \overline{v}' : S' \vdash \mathtt{ok}$ implies $E_1; v : \mathsf{Eq}(p : M); E_2 \vdash \mathtt{ok}$ and $E_1; v : \mathsf{Eq}(p : M); E_2; E_3 \vdash S' : \mathtt{spec}$. Let $E = E_1; v : \mathsf{Eq}(p : M); E_2; E_3$ and $E' = E_1; v : \mathsf{Ty}(M/p); E_2; E_3$. By induction hypothesis we have $E' \vdash \mathtt{ok}$.

Since S' is a term specification $S' = \mathsf{Ty}(t)$ or $\mathsf{Eq}(e : t)$. By inversion we have $E \vdash t : \mathrm{s}$ and $E \vdash e : t$ in case of a manifest specification. In either case we can use Lemmas 5.8.7 and 8.2.1 to get $E' \vdash t : \mathrm{s}$ or $E' \vdash e : t$ respectively. Now we can use **(Ty/Form)** or **(Eq/Form)** to obtain $E' \vdash S' : \mathtt{spec}$ and **(Env/Insert)** to conclude with $E'; v' : S' \vdash \mathtt{ok}$. □

The lemma below corresponds to Lemmas 7.1.2 and 7.1.4. In fact it is stronger than the latter two, since they are valid only for a classic E_2 (i.e. without modules and without manifest blocks).

Lemma 8.2.3. *Let $E_1; v : \mathsf{Eq}(p : M); E_2 \vdash \mathtt{ok}$ where E_2 is an environment without modules. Then $E_1; v : \mathsf{Ty}(M/p); E_2 \vdash \mathtt{ok}$.*

Proof. By induction on the length of E_2. If it is empty, then $E_1 \vdash p : M$ and hence $E_1 \vdash M/p : \mathtt{modtype}$ and $E_1, v : \mathsf{Ty}(M/p) \vdash \mathtt{ok}$.

Otherwise $E_1; v : \mathsf{Eq}(p : M); E_2; \overline{v}' : S' \vdash \mathtt{ok}$ implies $E_1; v : \mathsf{Eq}(p : M); E_2 \vdash \mathtt{ok}$ and $E_1; v : \mathsf{Eq}(p : M); E_2 \vdash S' : \mathtt{spec}$. Let $E = E_1; v : \mathsf{Eq}(p : M); E_2$ and $E' = E_1; v : \mathsf{Ty}(M/p); E_2$. By induction hypothesis we have $E' \vdash \mathtt{ok}$. Since S' is not a module specification, it is either a term specification or a block specification.

If S' is a term specification then we can use Lemma 8.2.2,

Otherwise S' is a block specification $S' = \mathsf{Ty}(BT)$ or $\mathsf{Eq}(\overline{p} : BT)$. By inversion, we have $E \vdash BT : \texttt{blocktype}$ and $E \vdash \overline{p} : BT$ in case of a manifest specification.

We will first show that $E' \vdash BT : \texttt{blocktype}$. By Lemma 5.8.7 we get the validity of judgments in $\mathit{Essence}_P(E \vdash BT : \texttt{blocktype})$. Let us call $\mathcal{J}$ this set of judgments. Since by Lemma 5.8.1 every judgment in $\mathcal{J}$ is a term judgment of the form $E; E_3 \vdash J'$ where E_3 is a term environment, we can apply Lemma 8.2.2 to get $E'; E_3 \vdash \texttt{ok}$ and Lemma 8.2.1 to get $E'; E_3 \vdash J$

Let us call the resulting set of judgments $\mathcal{J}'$. Of course,

$$\mathcal{J}' \;=\; \{E'; E_3 \vdash J \;\mid\; E'; E_3 \vdash_P J \in \mathit{Essence}_P(E' \vdash BT : \texttt{blocktype})\}$$

Since the function $\mathit{Essence}_P(\cdot)$ for block correctness judgments is constructed as an inversion of the formation rule, we can use judgments in $\mathcal{J}'$ and the closure property (C5) to get $E' \vdash BT : \texttt{blocktype}$.

If S' is a manifest block specification $\mathsf{Eq}(\overline{p} : BT)$ then also have to show $E' \vdash \overline{p} : BT$. We have to consider the cases of $BT = \mathsf{RewSpec}(E^\Gamma, R)$ and $\mathsf{IndSpec}(E^I := E^C)$ separately.

In the first case, from $E \vdash \overline{p} : \mathsf{RewSpec}(E^\Gamma, R)$ we can deduce the validity of all $\vdash$ judgments corresponding to $\vdash_P$ judgments in $\mathit{Essence}_P(E' \vdash \overline{p} : \mathsf{RewSpec}(E^\Gamma, R))$. Since $E' \vdash \mathsf{RewSpec}(E^\Gamma, R) : \texttt{blocktype}$ we can use the rule **(RewSpec/Sat)** to get $E' \vdash \overline{p} : \mathsf{RewSpec}(E^\Gamma, R)$ itself.

In case $BT = \mathsf{IndSpec}(E^I := E^C)$ we have to use Lemma 8.1.5 applied to $E \vdash \overline{p} : \mathsf{IndSpec}(E^I := E^C)$ in order to get

$$E \vdash_P \overline{p} : \mathsf{IndSpec}(E^{I'} := E^{C'}) \quad \text{and} \quad E \vdash \mathsf{IndSpec}(E^I := E^C) \approx \mathsf{IndSpec}(E^{I'} := E^{C'})$$

From the first judgment we get $E' \vdash \overline{p} : \mathsf{IndSpec}(E^{I'} := E^{C'})$ by Lemma 8.2.1 and by Lemma 4.1.15 we also get $E' \vdash \mathsf{IndSpec}(E^{I'} := E^{C'}) : \texttt{blocktype}$.

From the convertibility judgment, we can get the validity of judgments in

$$\mathit{Essence}_P(E \vdash \mathsf{IndSpec}(E^I := E^C) \approx \mathsf{IndSpec}(E^{I'} := E^{C'}))$$

by Lemma 5.8.7 and hence the $\vdash$ judgments corresponding to $\vdash_P$ judgments from

$$\mathit{Essence}_P(E' \vdash \mathsf{IndSpec}(E^I := E^C) \approx \mathsf{IndSpec}(E^{I'} := E^{C'}))$$

by Lemmas 8.2.2 and 8.2.1. Using $E' \vdash \mathsf{IndSpec}(E^I\delta := E^C\delta) : \texttt{blocktype}$ and $E' \vdash \mathsf{IndSpec}(E^{I'} := E^{C'}) : \texttt{blocktype}$ and the latter set of judgment we can get

$$E' \vdash \mathsf{IndSpec}(E^I := E^C) \approx \mathsf{IndSpec}(E^{I'} := E^{C'})$$

by **(Conv/IndSpec)**.

Finally, using **(Block/Conv)** we get $E' \vdash \overline{p} : \mathsf{IndSpec}(E^I := E^C)$.

In both cases, for $BT = \mathsf{IndSpec}(E^I := E^C)$ and for $BT = \mathsf{RewSpec}(E^\Gamma, R)$, we get $E' \vdash \overline{p} : BT$ and $E' \vdash BT : \texttt{blocktype}$, so we can apply **(Eq/Form)** to get $E' \vdash \mathsf{Eq}(\overline{p} : BT) : \texttt{spec}$ and **(Env/Insert)** to get $E'; \overline{v} : \mathsf{Eq}(\overline{p} : BT) \vdash \texttt{ok}$. $\square$

8.3 Removing modules

The first two lemmas of this section close the gap between Lemma 8.2.3 and Lemmas 7.1.2 and 7.1.4. The remaining three show that module removal operations: signature flattening, functor removal and elimination of manifest modules are bidirectional, i.e. a term judgment is valid before the operation if and only if it is valid after the operation.

Lemma 8.3.1. *Let* $E_1; v : \mathsf{Ty}(M); E_2 \vdash \mathtt{ok}$ *where* E_2 *is an environment without modules and* $M = \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$. *Then* $E_1; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n; E_2\delta \vdash \mathtt{ok}$*, where* $\delta = \{\overline{v.\mathtt{v}_i} \mapsto \overline{v}_i\}$.

Proof. By induction on the length of E_2. If E_2 is empty then the result follows from the correctness of the signature M. Otherwise $E_1; v : \mathsf{Ty}(M); E_2; \overline{v}' : S' \vdash \mathtt{ok}$ implies $E_1; v : \mathsf{Ty}(M); E_2 \vdash \mathtt{ok}$ and $E_1; v : \mathsf{Ty}(M); E_2 \vdash S' : \mathtt{spec}$. Let $E = E_1; v : \mathsf{Ty}(M); E_2$ and $E' = E_1; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n; E_2\delta$. By induction hypothesis we have $E' \vdash \mathtt{ok}$. Since S' is not a module specification, it is either a term specification or a block specification.

If S' is a term specification we just use Lemma 7.1.1.

If S' is an abstract block judgment, we proceed in the same way as in the proof of Lemma 7.1.2. If S' is a manifest block specification $\mathsf{Eq}(\overline{p} : BT)$ then by inversion we have $E \vdash \overline{p} : BT$ and $E \vdash BT : \mathtt{blocktype}$. From the second judgment, we can deduce $E' \vdash BT\delta : \mathtt{blocktype}$, just as in the proof of Lemma 7.1.2.

Now we have to consider the cases of $BT = \mathsf{RewSpec}(E^\Gamma, R)$ and $\mathsf{IndSpec}(E^I := E^C)$ separately.

In the first case, from $E \vdash \overline{p} : \mathsf{RewSpec}(E^\Gamma, R)$ we can deduce the validity of all judgments in $Essence_P(E' \vdash \overline{p}\delta : \mathsf{RewSpec}(E^\Gamma\delta, R\delta))$. Since $E' \vdash \mathsf{RewSpec}(E^\Gamma\delta, R\delta) : \mathtt{blocktype}$ we can use the rule **(RewSpec/Sat)** to get $E' \vdash \overline{p}\delta : \mathsf{RewSpec}(E^\Gamma\delta, R\delta)$ itself.

In case $BT = \mathsf{IndSpec}(E^I := E^C)$ we have to use Lemma 8.1.5 applied to $E \vdash \overline{p} : \mathsf{IndSpec}(E^I := E^C)$ in order to get

$$E \vdash_P \overline{p} : \mathsf{IndSpec}(E^{I'} := E^{C'}) \quad \text{and} \quad E \vdash \mathsf{IndSpec}(E^I := E^C) \approx \mathsf{IndSpec}(E^{I'} := E^{C'})$$

From the first judgment we get $E' \vdash_P \overline{p}\delta : \mathsf{IndSpec}(E^{I'}\delta := E^{C'}\delta)$ by Lemma 5.9.1 and by Lemma 4.1.15 we also get $E' \vdash \mathsf{IndSpec}(E^{I'}\delta := E^{C'}\delta) : \mathtt{blocktype}$.

From the convertibility judgment, we can get the validity of judgments in

$$Essence_P(E \vdash \mathsf{IndSpec}(E^I := E^C) \approx \mathsf{IndSpec}(E^{I'} := E^{C'}))$$

by Lemma 5.8.7 and hence in

$$Essence_P(E' \vdash \mathsf{IndSpec}(E^I\delta := E^C\delta) \approx \mathsf{IndSpec}(E^{I'}\delta := E^{C'}\delta))$$

by Lemmas 7.1.1 and 5.9.1. Using $E' \vdash \mathsf{IndSpec}(E^I\delta := E^C\delta) : \mathtt{blocktype}$ and $E' \vdash \mathsf{IndSpec}(E^{I'}\delta := E^{C'}\delta) : \mathtt{blocktype}$ and the latter set of judgment we can get

$$E' \vdash \mathsf{IndSpec}(E^I\delta := E^C\delta) \approx \mathsf{IndSpec}(E^{I'}\delta := E^{C'}\delta)$$

by **(Conv/IndSpec)**.

Finally, using **(Block/Conv)** we get $E' \vdash \overline{p}\delta : \mathsf{IndSpec}(E^I\delta := E^C\delta)$.

In both cases, for $BT = \mathsf{IndSpec}(E^I := E^C)$ and for $BT = \mathsf{RewSpec}(E^\Gamma, R)$, we get $E' \vdash \overline{p}\delta : BT\delta$ and $E' \vdash BT\delta : \mathtt{blocktype}$, so we can apply **(Eq/Form)** to get $E' \vdash \mathsf{Eq}(\overline{p}\delta : BT\delta) : \mathtt{spec}$ and **(Env/Insert)** to get $E'; \overline{v} : \mathsf{Eq}(\overline{p}\delta : BT\delta) \vdash \mathtt{ok}$. ☐

Lemma 8.3.2. *Let $E_1; v : \mathsf{Ty}(M); E_2 \vdash \mathtt{ok}$ where E_2 is an environment without modules and $M = \mathsf{Funsig}(\hat{v} : M_1)M_2$. Then $E_1; E_2 \vdash \mathtt{ok}$.*

Proof. The proof is similar to the proof of Lemma 8.3.1. ☐

Lemma 8.3.3. *Let $E = E_1; v : \mathsf{Ty}(M); E_2$, where E_2 is an environment without modules and $M = \mathsf{Sig}\ \overline{\mathtt{v}}_1 \triangleright \overline{v}_1 : S_1 \ldots \overline{\mathtt{v}}_n \triangleright \overline{v}_n : S_n\ \mathsf{End}$. If $E \vdash \mathtt{ok}$ then for every term judgment J,*

$$E \vdash J \quad \textit{if and only if} \quad E' \vdash J\delta$$

where $E' = E_1; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n; E_2\delta$ and $\delta = \{\overline{v.\mathtt{v}_i} \mapsto \overline{v}_i\}$.

Proof. The proof of the "only if" part is almost already done. If $E \vdash J$ then $E \vdash_P J$ by Lemma 5.8.7 and since $E' \vdash \mathtt{ok}$ by Lemma 8.3.1, we can apply Lemma 5.9.1 to get $E' \vdash J\delta$.

The proof of the "if" part is also quite simple. If $E' \vdash J\delta$ then by Lemma 4.1.4,

$$E_1; v : \mathsf{Ty}(M); \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n; E_2\delta \vdash J\delta$$

Now, since $E_1; v : \mathsf{Ty}(M) \vdash \overline{v.\mathtt{v}_1} : S_1$, by Lemma 4.1.5

$$E_1; v : \mathsf{Ty}(M); \overline{v}_2 : S_2\{\overline{v}_1 \mapsto \overline{v.\mathtt{v}_1}\} \ldots \overline{v}_n : S_n\{\overline{v}_1 \mapsto \overline{v.\mathtt{v}_1}\}; E_2\delta\{\overline{v}_1 \mapsto \overline{v.\mathtt{v}_1}\} \vdash J\delta\{\overline{v}_1 \mapsto \overline{v.\mathtt{v}_1}\}$$

Now we have $E_1; v : \mathsf{Ty}(M) \vdash \overline{v.\mathtt{v}_2} : S_2\{\overline{v}_1 \mapsto \overline{v.\mathtt{v}_1}\}$ and we can continue substituting $\overline{v.\mathtt{v}_j}$ for $\overline{v}_j$ until $j = n$. In the end, we obtain

$$E_1; v : \mathsf{Ty}(M); E_2\delta\{\overline{v}_i \mapsto \overline{v.\mathtt{v}_i}\} \vdash J\delta\{\overline{v}_i \mapsto \overline{v.\mathtt{v}_i}\}$$

Since $\delta\{\overline{v}_i \mapsto \overline{v.\mathtt{v}_i}\}$ is the identity, the latter is equivalent $E \vdash J$, which concludes the proof. ☐

Lemma 8.3.4. *Let $E = E_1; v : \mathsf{Ty}(M); E_2$, where E_2 is an environment without modules and $M = \mathsf{Funsig}(\hat{v} : M_1)M_2$. If $E \vdash \mathtt{ok}$ then for every term judgment J,*

$$E \vdash J \quad \textit{if and only if} \quad E' \vdash J$$

where $E' = E_1; E_2$.

Proof. The proof of the "only if" part is the following: $E \vdash J$ implies $E \vdash_P J$ and since $E' \vdash \mathtt{ok}$ by Lemma 8.3.2, we have $E' \vdash J$ by Lemma 5.9.2.

The proof of the "if" part simply follows by Lemma 4.1.4. ☐

Lemma 8.3.5. *Let $E = E_1; v : \mathsf{Eq}(p : M); E_2$, where E_2 is an environment without modules. If $E \vdash \mathtt{ok}$ then for every term judgment J,*

$$E \vdash J \quad \textit{if and only if} \quad E' \vdash J$$

where $E' = E_1; v : \mathsf{Ty}(M/p); E_2$.

Proof. The proof of the "only if" part is the following: $E \vdash J$ implies $E \vdash_P J$ and since $E' \vdash \mathtt{ok}$ by Lemma 8.2.3, we have $E' \vdash J$ by Lemma 8.2.1.

The proof of the "if" part is a bit more involved. If $E' \vdash J$ then by Lemma 4.1.4,

$$E_1; v' : \mathsf{Eq}(p : M); v : \mathsf{Ty}(M/p); E_2 \vdash J$$

Since $E_1; v' : \mathsf{Eq}(p : M) \vdash v' : M$ and $E_1; v' : \mathsf{Eq}(p : M) \vdash p : M$ and $E_1; v' : \mathsf{Eq}(p : M) \vdash v' \approx p$, we get $E_1; v' : \mathsf{Eq}(p : M) \vdash M/v' <: M/p$ by Lemma 5.6.1.

Hence $E_1; v' : \mathsf{Eq}(p : M) \vdash v' : M/p$ and consequently $E_1; v' : \mathsf{Eq}(p : M) \vdash v' : \mathsf{Ty}(M/p)$. By Lemma 4.1.5 we get

$$E_1; v' : \mathsf{Eq}(p : M); E_2\{v \mapsto v'\} \vdash J\{v \mapsto v'\}$$

which is α-convertible with the desired conclusion $E_1; v : \mathsf{Eq}(p : M); E_2 \vdash J$. □

8.4 Removing manifest blocks

In this section we prove that the operation of removing manifest blocks is bidirectional (Lemma 8.4.5), i.e. given an environment $E = E_1; \overline{v} : \mathsf{Eq}(\overline{v}' : BT); E_2$ without modules and $E' = E_1; E_2\delta$ where $\delta = \{\overline{v} \mapsto \overline{v}'\}$, we prove that $E \vdash e : t$ if and only if $E' \vdash e\delta : t\delta$. The proof of the "only if" part is trivial, but proving the "if" part is more complicated. First, we need Lemma 8.4.4 to show that $E' \vdash e\delta : t\delta$ implies $E \vdash e : t\delta$ and then Lemma 8.4.2 to get $E \vdash t \approx t\delta$ and hence $E \vdash e : t$ by conversion.

Lemma 8.4.4 is quite complex and it needs a number of auxiliary facts. We start by proving that the environment with a manifest block subsumes the environment without the manifest block.

Lemma 8.4.1. *Let $E_1; \overline{v} : \mathsf{Eq}(\overline{\tilde{v}} : BT); E_2 \vdash \mathtt{ok}$ where E_1 and E_2 are environments without modules and $\delta = \{\overline{v} \mapsto \overline{\tilde{v}}\}$. If $E_1; E_2\delta \vdash J$ then $E_1; \overline{v} : \mathsf{Eq}(\overline{\tilde{v}} : BT); E_2 \vdash J$.*

Proof. We prove by induction on the length of E_2 that the implication holds for all judgments J. If E_2 is empty then it is trivial by Lemma 4.1.4. Otherwise $E_1; \overline{v} : \mathsf{Eq}(\overline{\tilde{v}} : BT); E_2; \overline{v}' : S' \vdash \mathtt{ok}$ implies $E_1; \overline{v} : \mathsf{Eq}(\overline{\tilde{v}} : BT); E_2 \vdash \mathtt{ok}$ and $E_1; \overline{v} : \mathsf{Eq}(\overline{\tilde{v}} : BT); E_2 \vdash S' : \mathtt{spec}$. Let $E = E_1; \overline{v} : \mathsf{Eq}(\overline{\tilde{v}} : BT); E_2$ and $E' = E_1; E_2\delta$.

By induction hypothesis if $E' \vdash J$ then $E \vdash J$ for any judgment J.

Since $E'; \overline{v}' : S'\delta \vdash \mathtt{ok}$ we have $E' \vdash S'\delta : \mathtt{spec}$. By induction hypothesis, we also have $E \vdash S'\delta : \mathtt{spec}$. Since $E \vdash S' : \mathtt{spec}$ and $E \vdash \overline{v} \approx \overline{\tilde{v}}$, by Lemma 4.1.13 we get $E \vdash S' \approx S'\delta$. By Lemma 4.1.10 we get $E; \overline{v} : S' \vdash \overline{v} : S'\delta$. Consequently, $E; \overline{v} : S' \vdash \overline{v}'' : S''$ for every $\overline{v}'' : S''$ in $E'; \overline{v} : S'\delta$ and hence by Lemma 4.1.6 we get $E; \overline{v} : S' \vdash J$ for every J such that $E'; \overline{v} : S'\delta \vdash J$. □

Lemma 8.4.2. *Let $E = E_1; \overline{v} : \mathsf{Eq}(\overline{v}' : BT); E_2$ be an environment without modules and let $\delta = \{\overline{v} \mapsto \overline{v}'\}$. If $E \vdash \phi : \Psi$ then $E \vdash \phi \approx \phi\delta$.*

Proof. If $E \vdash \phi : \Psi$ then $E' \vdash \phi\delta : \Psi\delta$ where $E' = E_1; E_2\delta$. By Lemma 8.4.1 $E \vdash \phi\delta : \Psi\delta$ and since $E \vdash \overline{v} \approx \overline{v}'$, we have $E \vdash \phi \approx \phi\delta$ by Lemma 4.1.13. □

Lemma 8.4.4, the most complicated lemma of this section, requires a long proof by induction on the derivation in $\vdash_P$, using the function $Essence_P(\cdot)$ defined in Section 5.8 and the notion of direct type defined below.

Definition 8.4.1. *Given an environment without modules E and a variable v, the* direct type *of v in E, denoted by $E(v)$ is defined as follows:*

- *if $E = E_1; v : S; E_2$ where $S = \mathsf{Ty}(t)$ or $\mathsf{Eq}(e : t)$ then $E(v) = t$*
- *if $E = E_1; \overline{v} : S; E_2$ where $\overline{v} = v^1 \dots v^n$ and $v = v^i$, $S = \mathsf{Ty}(BT)$ or $\mathsf{Eq}(\overline{\tilde{v}} : BT)$, then:*
  - *if $BT = \mathsf{RewSpec}(v'_1 : \mathsf{Ty}(t_1) \dots v'_n : \mathsf{Ty}(t_n), R)$ then $E(v) = t_i$*
  - *if $BT = \mathsf{IndSpec}(v^I_1 : \mathsf{Ty}(t_1) \dots v^I_k : \mathsf{Ty}(t_k) := E^C)$ and $i \leq k$ then $E(v) = t_i$*
  - *if $BT = \mathsf{IndSpec}(v^I_1 : \mathsf{Ty}(t_1) \dots v^I_k : \mathsf{Ty}(t_k) := v^C_1 : \mathsf{Ty}(\hat{t_1}) \dots v^C_l : \mathsf{Ty}(\hat{t_l}))$ and $i > k$ then $E(v) = \hat{t}_{i-k}\{v^I_j \mapsto v^j\}$*

Lemma 8.4.3. *Let E be an environment without modules. For every v in the domain of E, $E \vdash_P v : E(v)$. Moreover if $E \vdash v : t$ then $E \vdash_P t \approx E(v)$.*

Proof. The judgment $E \vdash_P v : E(v)$ can be derived using the rule **(P/Env/Lookup)**, followed by either **(P/Ty/Type)** or **(P/Eq/Type)**, followed by **(P/RewSpec/Fun)** or **(P/IndSpec/IndType)** or **(P/IndSpec/Constr)** in case v is a part of a block.

The second part follows from the fact that every term judgment can be derived in $\vdash_P$ and that the above derivation of $E \vdash_P v : E(v)$ is the only one in $\vdash_P$ which makes v appear on the left-hand side of : in the judgment $E \vdash_P v : t$. The only rule that does not change that is **(P/Term/Conv)** and therefore $E \vdash_P t \approx E(v)$ by transitivity. □

Lemma 8.4.4. *Let $E = E_1; \overline{v} : \mathsf{Eq}(\overline{v}' : BT); E_2$ be an environment without modules, and let $E' = E_1; E_2\delta$ where $\delta = \{\overline{v} \mapsto \overline{v}'\}$. If $E \vdash \mathtt{ok}$ and $E' \vdash_P \phi\delta : \Psi$ then all judgments in $Essence_P(E \vdash \phi : \Psi)$ are satisfied.*

Proof. We start by proving that $E \vdash v^j : E(v^{j\prime})$ for every $v^j \in \overline{v}$. We have to consider two cases.

- If $BT = \mathsf{RewSpec}(E^\Gamma, R)$ with $E^\Gamma = v^\Gamma_1 : \mathsf{Ty}(t_1) \dots v^\Gamma_n : \mathsf{Ty}(t_n)$ or $BT = \mathsf{IndSpec}(E^I := E^C)$ with $E^I = v^I_1 : \mathsf{Ty}(t_1) \dots v^I_k : \mathsf{Ty}(t_k)$ and $j \leq k$, then $E_1 \vdash v^{j\prime} : t_j$ and by Lemma 8.4.3 and Lemma 5.7.2, $E \vdash_P t_j \approx E(v^{j\prime})$.

  For v^j we have $E(v^j) = t_j$ and hence $E \vdash v^j : t_j$. Using **(Term/Conv)**, we obtain $E \vdash v^j : E(v^{j\prime})$.

- Otherwise $BT = \mathsf{IndSpec}(E^I := E^C)$ with $E^I = v_1^I : \mathsf{Ty}(t_1) \dots v_k^I : \mathsf{Ty}(t_k)$ and $E^C = v_1^C : \mathsf{Ty}(\hat{t}_1) \dots v_l^C : \mathsf{Ty}(\hat{t}_l)$ and $j > k$. In that case $E_1 \vdash v^{j\prime} : \hat{t}_{j-k}\{v_i^I \mapsto v^{i\prime}\}$ and $E \vdash_P \hat{t}_{j-k}\{v_i^I \mapsto v^{i\prime}\} \approx E(v^{j\prime})$.

  For v^j we have $E(v^j) = \hat{t}_{j-k}\{v_i^I \mapsto v^i\}$ but fortunately $E \vdash v^i \approx v^{i\prime}$ for all i and we can use Lemma 4.1.13 to get $E \vdash \hat{t}_{j-k}\{v_i^I \mapsto v^{i\prime}\} \approx \hat{t}_{j-k}\{v_i^I \mapsto v^i\}$. By transitivity and conversion, we obtain $E \vdash v^j : E(v^{j\prime})$.

Now we can do the proof by induction on the derivation of $E' \vdash_P \phi\delta : \Psi$. The most difficult is the rule **(P/Env/Lookup)**. The conclusion of this rule is $E' \vdash_P \overline{\overline{v}}\delta : S$.

If $\overline{\overline{v}}\delta = \overline{\overline{v}}$ then by Lemma 8.4.1, $E \vdash \overline{\overline{v}} : S$.

Otherwise $\overline{\overline{v}} \cap \overline{v} \neq \emptyset$ and hence $\overline{\overline{v}}\delta \cap \overline{v}' \neq \emptyset$. In other words $\overline{\overline{v}}\delta : S$ is a part of E_1. We consider the following cases, depending on S.

- If $S = \mathsf{Ty}(t)$, then $\overline{\overline{v}}\delta$ has length 1 and $\overline{\overline{v}}\delta = v^{j\prime}$ for some j and consequently $\overline{\overline{v}} = v^j$. Since $t = E(v^{j\prime})$, we already know that $E \vdash v^j : t$.

- If $S = \mathsf{Eq}(e : t)$, then $\overline{\overline{v}} = v^j$ for some j as in the previous case and also $E \vdash v^j : t$. In order to prove $E \vdash v^j \approx e$ it is enough to notice that $E \vdash v^j \approx v^{j\prime}$ and $E \vdash v^{j\prime} \approx e$.

- If $S = \mathsf{Ty}(\mathsf{RewSpec}(E^\Gamma, R))$ where $E^\Gamma = v_1^\Gamma : \mathsf{Ty}(t_1) \dots v_n^\Gamma : \mathsf{Ty}(t_n)$ and the set of rules $R = \{E_i \mid e_i^L \longrightarrow e_i^R : \hat{t}_i\}_{i=1\dots m}$ then $\overline{\overline{v}}$ has length n. We have

$$\begin{aligned} &\mathit{Essence}_P(E \vdash \overline{\overline{v}} : S) = \\ &\qquad \{E \vdash_P \tilde{v}^j : t_j \ \text{ for } j = 1 \dots n\} \quad \cup \\ &\qquad \{E; E_i \vdash_P e_i^L\{v_j^\Gamma \mapsto \tilde{v}^j\} \approx e_i^R\{v_j^\Gamma \mapsto \tilde{v}^j\} \ \text{ for } i = 1 \dots m\} \end{aligned}$$

  By assumption we have $E' \vdash \tilde{v}^j\delta : t_j$ for $j = 1 \dots n$ and

$$E'; E_i \vdash e_i^L\{v_j^\Gamma \mapsto \tilde{v}^j\delta\} \approx e_i^R\{v_j^\Gamma \mapsto \tilde{v}^j\delta\} \ \text{ for } i = 1 \dots m$$

  If, for a given j $\tilde{v}^j\delta = \tilde{v}^j$ then by Lemma 8.4.1, we have $E \vdash \tilde{v}^j : t_j$. Otherwise, if $\tilde{v}^j = v^i$ and $\tilde{v}^j\delta = v^{i\prime}$ then we have $E \vdash v^i : t_j$, because $t_j = E(v^{i\prime})$.

  For convertibility judgments, it is enough to notice that by inverting the judgment $E' \vdash \mathsf{RewSpec}(E^\Gamma, R) : \mathtt{blocktype}$, substitution and Lemma 8.4.1, we get

$$E; E_i \vdash e_i^L\{v_j^\Gamma \mapsto \tilde{v}^j\delta\} : \hat{t}_i\{v_j^\Gamma \mapsto \tilde{v}^j\delta\} \ \text{ and } \ E; E_i \vdash e_i^R\{v_j^\Gamma \mapsto \tilde{v}^j\delta\} : \hat{t}_i\{v_j^\Gamma \mapsto \tilde{v}^j\delta\}$$

  Similarly, by inverting $E \vdash \mathsf{RewSpec}(E^\Gamma, R) : \mathtt{blocktype}$ and substitution we get

$$E; E_i \vdash e_i^L\{v_j^\Gamma \mapsto \tilde{v}^j\} : \hat{t}_i\{v_j^\Gamma \mapsto \tilde{v}^j\} \ \text{ and } \ E; E_i \vdash e_i^R\{v_j^\Gamma \mapsto \tilde{v}^j\} : \hat{t}_i\{v_j^\Gamma \mapsto \tilde{v}^j\}$$

  Since $E \vdash \tilde{v}^j \approx \tilde{v}^j\delta$, we can apply Lemma 4.1.13 to get

$$E; E_i \vdash e_i^L\{v_j^\Gamma \mapsto \tilde{v}^j\delta\} \approx e_i^L\{v_j^\Gamma \mapsto \tilde{v}^j\} \ \text{ and } \ E; E_i \vdash e_i^R\{v_j^\Gamma \mapsto \tilde{v}^j\delta\} \approx e_i^R\{v_j^\Gamma \mapsto \tilde{v}^j\}$$

  This, by Lemma 8.4.1 and transitivity gives

$$E; E_i \vdash e_i^L\{v_j^\Gamma \mapsto \tilde{v}^j\} \approx e_i^R\{v_j^\Gamma \mapsto \tilde{v}^j\}$$

- If $S = \mathsf{Ty}(\mathsf{IndSpec}(E^I := E^C))$ where $E^I = v_1^I : \mathsf{Ty}(t_1) \ldots v_k^I : \mathsf{Ty}(t_k)$ and $E^C = v_1^C : \mathsf{Ty}(\hat{t}_1) \ldots v_n^C : \mathsf{Ty}(\hat{t}_n)$ then the $length(\overline{\tilde{v}}) = k + l$ we have to show the validity of judgments in

  $Essence_P(E \vdash \overline{\tilde{v}} : S) =$
  $$\{E \vdash_P \tilde{v}^j : t_j \ \text{ for } j = 1 \ldots k\} \quad \cup \quad \{E \vdash_P \tilde{v}^{k+i} : \hat{t}_i\{v_j^I \mapsto \tilde{v}^j\} \ \text{ for } i = 1 \ldots n\}$$

  The first set of judgments can be obtained in the same way as in the previous case. For the second set, by hypothesis we have $E \vdash \tilde{v}^{k+i}\delta : \hat{t}_i\{v_j^I \mapsto \tilde{v}^j\delta\}$. Just as in the previous case it can be shown that

  $$E \vdash \hat{t}_i\{v_j^I \mapsto \tilde{v}^j\delta\} \approx \hat{t}_i\{v_j^I \mapsto \tilde{v}^j\}$$

  using Lemma 4.1.13 since both substitutions are convertible and both types are correct.

  Now if $\tilde{v}^{k+i}\delta = \tilde{v}^{k+i}$ then by conversion $E \vdash \tilde{v}^{k+i} : \hat{t}_i\{v_j^I \mapsto \tilde{v}^j\}$.

  Otherwise if $\tilde{v}^{k+i} = \tilde{v}^l$ and $\tilde{v}^{k+i}\delta = \tilde{v}^{l\prime}$ then $E \vdash \tilde{v}^l : \hat{t}_i\{v_j^I \mapsto \tilde{v}^j\delta\}$, because we have $\hat{t}_i\{v_j^I \mapsto \tilde{v}^j\delta\} = E(\tilde{v}^{l\prime})$ and by conversion we get $E \vdash \tilde{v}^l : \hat{t}_i\{v_j^I \mapsto \tilde{v}^j\}$.

- If $S = \mathsf{Eq}(\overline{\tilde{v}} : BT)$, then

  $$Essence_P(E \vdash \overline{\tilde{v}} : S) \ = \ \{E \vdash \tilde{v}^j \approx \hat{v}^j \ \text{ for all } j\} \cup Essence_P(E \vdash \overline{\tilde{v}} : BT)$$

  The first part can be shown as in the case for manifest term specifications and the second part is equal $Essence_P(E \vdash \overline{\tilde{v}} : \mathsf{Ty}(BT))$ and can by shown like in one of the two previous cases.

So we are done with the rule **(P/Env/Lookup)**. Other rules are much simpler to handle.

The rule **(P/Sig/Access)** is impossible in the environment without modules.

The rules for specifications all follow by induction hypothesis since $Essence_P(\cdot)$ applied to the conclusion is included in $Essence_P(\cdot)$ of the premises.

The term rules have to be treated with care. The rule **(P/Term/Prod)** in our case is the following.

$$\frac{E' \vdash_P t_1\delta : \mathrm{s}_1 \qquad E'; v : \mathsf{Ty}(t_1\delta) \vdash_P t_2\delta : \mathrm{s}_2}{E' \vdash_P (v{:}t_1\delta)t_2\delta : \mathrm{s}_3} \quad \text{if } (\mathrm{s}_1, \mathrm{s}_2, \mathrm{s}_3) \in \mathcal{R}$$

By the first induction hypothesis, we get $E \vdash_P t_1 : \mathrm{s}_1$, and since $E; v : \mathsf{Ty}(t_1) \vdash \mathtt{ok}$, by the second induction hypothesis, we get $E; v : \mathsf{Ty}(t_1) \vdash_P t_2 : \mathrm{s}_2$. Hence we can apply **(P/Term/Prod)** and get $E \vdash_P (v{:}t_1)t_2 : \mathrm{s}_3$ as desired.

The rule **(P/Term/Abs)** is the following:

$$\frac{E'; v : \mathsf{Ty}(t_1\delta) \vdash_P e\delta : t_2 \qquad E' \vdash_P (v{:}t_1\delta)t_2 : \mathrm{s}}{E' \vdash_P [v{:}t_1\delta]e\delta : (v{:}t_1\delta)t_2}$$

By Lemma 5.7.3, the second premise must contain the subderivation for product with the premises $E' \vdash_P t_1\delta : \mathrm{s}_1$ and $E'; v : \mathsf{Ty}(t_1\delta) \vdash_P t_2 : \mathrm{s}_2$ and the conclusion $E' \vdash_P (v{:}t_1\delta)t_2 : \mathrm{s}_3$ where $E' \vdash \mathrm{s}_3 \approx \mathrm{s}$ or $\mathrm{s}_3 = \mathrm{s}$. So by induction hypothesis we get $E \vdash t_1 : \mathrm{s}_1$ and hence $E; v : \mathsf{Ty}(t_1) \vdash \mathtt{ok}$. Using Lemma 8.4.1 we get $E \vdash t_1\delta : \mathrm{s}_1$ and $E; v : \mathsf{Ty}(t_1) \vdash t_2 : \mathrm{s}_2$ and by **(Term/Prod)** we get $E \vdash (v : t_1)t_2 : \mathrm{s}_3$.

Using Lemma 8.4.2, we get $E \vdash t_1 \approx t_1\delta$ and consequently $E; v : \mathsf{Ty}(t_1\delta) \vdash t_2 : \mathrm{s}_2$ and $E \vdash (v : t_1\delta)t_2 : \mathrm{s}_3$ by **(Term/Prod)**. Using **(Conv/Prod)** we can show that $E \vdash (v : t_1)t_2 \approx (v : t_1\delta)t_2$.

Now we can use induction hypothesis applied to the first premise to get $E; v : \mathsf{Ty}(t_1) \vdash e : t_2$ and apply **(Term/Abs)** to get $E \vdash [v{:}t_1]e : (v{:}t_1)t_2$. By conversion, we get $E \vdash [v{:}t_1]e : (v{:}t_1\delta)t_2$ as desired.

The rule **(P/Term/Axiom)** is obvious.

The rule **(P/Term/App)** looks as follows:

$$\frac{E' \vdash_P e\delta : (v{:}t_1)t_2 \qquad E' \vdash_P e'\delta : t_1}{E' \vdash_P e\delta\ e'\delta : t_2\{v \mapsto e'\delta\}}$$

By induction hypothesis we get $E \vdash e : (v{:}t_1)t_2$ and $E \vdash e' : t_1$, so we can apply **(Term/App)** and get $E \vdash e\,e' : t_2\{v \mapsto e'\}$.

Fortunately, by Lemma 8.4.2, $E \vdash e' \approx e'\delta$ and since both $t_2\{v \mapsto e'\}$ and $t_2\{v \mapsto e'\delta\}$ are correct by Lemma 5.10.1, we get $E \vdash t_2\{v \mapsto e'\} \approx t_2\{v \mapsto e'\delta\}$ by Lemma 4.1.13.

Hence by conversion we get $E \vdash e\,e' : t_2\{v \mapsto e'\delta\}$.

The rule **(P/Term/Conv)** is the following:

$$\frac{E' \vdash_P e\delta : t \qquad E' \vdash_P t' : \mathrm{s} \qquad E' \vdash_P t \approx t'}{E' \vdash_P e\delta : t'}$$

By the first induction hypothesis, we get $E \vdash e : t$ and by Lemma 8.4.1 we get $E \vdash t' : \mathrm{s}$ and $E \vdash t \approx t'$. We can therefore apply conversion to get $E \vdash e : t'$ as desired.

The block rules simply follow by induction hypothesis. $\square$

Lemma 8.4.5. *Let $E = E_1; \overline{v} : \mathsf{Eq}(\overline{v}' : BT); E_2$ be an environment without modules, and let $E \vdash \mathtt{ok}$. Moreover let $E' = E_1; E_2\delta$ where $\delta = \{\overline{v} \mapsto \overline{v}'\}$.*

- *If $E' \vdash e\delta : t'$ then $E \vdash e : t'$.*
- *$E \vdash e : t$ if and only if $E' \vdash e\delta : t\delta$.*
- *If $E \vdash e_1 : t_1$ and $E \vdash e_2 : t_2$ then $E \vdash e_1 \approx e_2$ iff $E' \vdash e_1\delta \approx e_2\delta$.*

Proof. The first implication follows directly from Lemma 8.4.4.

The second statement is true, because the "only if" part follows from Lemma 4.1.5 and the "if" part can be proved using the first implication and Lemmas 8.4.2 and 5.10.1. Indeed, in case t is a sort, we have $t = t\delta$ and we get the result immediately by the first implication. Otherwise, by Lemma 5.10.1 we get $E' \vdash t\delta : \mathrm{s}$ and hence $E \vdash t : \mathrm{s}$ by the first implication. Now we can use Lemma 8.4.2 and get $E \vdash t\delta \approx t$. Using the first implication and the rule **(Term/Conv)** we obtain the desired conclusion $E \vdash e : t$.

The "only if" part of the third statement follows again by Lemma 4.1.5 and the "if" part can be proved using Lemma 8.4.2 and transitivity. $\square$

8.5 Relative decidability

Given the partial results proved in earlier sections of this chapter, we can state and prove our relative decidability result. In the decidability theorem we make the additional assumption that the side-conditions for formation rules for inductive blocks and rewriting blocks are decidable.

The two lemmas show two steps of our reduction of typability: first we remove module elements from the environment, and second, we remove manifest blocks.

Lemma 8.5.1. *Any environment E and a term judgment J can be transformed deterministically in polynomial time into an environment E' without modules and a term judgment J' such that*

$$E \vdash J \qquad \textit{if and only if} \qquad E' \vdash J'$$

Proof. It is enough to apply Lemmas 8.3.3, 8.3.4 and 8.3.5 until there is no more module components in the environment.

This procedure takes time polynomial in the weight of E and J. Indeed every abstract module removal has at most quadratic cost: for functor removal it is obvious and for signature flattening, the size of the substitution is linear, and performing a substitution is a traversal of the environment and the judgment, giving at most quadratic time. Every module removal reduces the weight, so their total cost is limited by the weight of E to the power 3.

Transforming a manifest module specification into an abstract one also takes linear time, does not reduce weight, but does not increase it either. Since the module component becomes abstract, it gets removed immediately afterwards. Therefore this operation can be performed at most once for every specification, its total time is limited by the weight of E to the power 2. $\square$

The preceding lemma is very important. It proves the correctness of the term typing algorithm implemented in Coq (see Section 9.2.4).

Lemma 8.5.2. *Any environment E without modules and a term judgment J can be transformed deterministically in polynomial time into a classic environment E' and a term judgment J' such that*

$$E \vdash J \qquad \textit{if and only if} \qquad E' \vdash J'$$

Proof. It is enough to apply Lemma 8.4.5 until there is no more manifest blocks in the environment. Every manifest block removal takes time quadratic in the size of the environment plus the size of the judgment.

When all manifest blocks are removed from the environment, we arrive at a classic environment. □

Therefore type-checking terms in the calculus with modules reduces in polynomial time to type-checking terms in the classic environments in the system $\vdash_P$. We can easily extend this result to all expressions of our language.

Theorem 2. *Type-checking in $\vdash$ is decidable if and only if type-checking in $\vdash_P$ in a classic environment is decidable.*

Proof. Lemmas 8.5.1 and 8.5.2 above show that type-checking terms in $\vdash$ reduces to type-checking in $\vdash_P$ in a classic environments.

Type-checking blocks reduces to type-checking terms as well. Indeed, by Lemma 8.1.5 a judgment of the form $E \vdash \overline{p} : \mathsf{IndSpec}(E^I := E^C)$ implies that $E \vdash_P \overline{p} : \mathsf{IndSpec}(E^{I'} := E^{C'})$ and $E \vdash \mathsf{IndSpec}(E^I := E^C) \approx \mathsf{IndSpec}(E^{I'} := E^{C'})$ for some $E^{I'}$ and $E^{C'}$. Finding suitable $E^{I'}$ and $E^{C'}$ is decidable, because type-checking blocks in $\vdash_P$ is straightforward. The only applicable rules are **(P/Env/Lookup)**, **(P/Sig/Access)**, **(Ty/Type)** and **(Eq/Type)** and one just has to follow the structure of the paths $\overline{p}$. Once $E^{I'}$ and $E^{C'}$ found, the second judgment can only be proved using the rule **(Conv/IndSpec)**, which has only term premises and block correctness judgments, which again follow from some term judgments.

For rewriting blocks, $E \vdash \overline{p} : \mathsf{RewSpec}(E^\Gamma, R)$ can be proved from $\mathit{Essence}_P(E \vdash \overline{p} : \mathsf{RewSpec}(E^\Gamma, R))$ and the latter contains only term judgments.

Decidability of modules comes from the principal types property, which is expressed for paths by Lemma 5.6.5 and for other module expressions by Lemma 4.2.1. The usual typing algorithm [62] may be applied which computes the principal type for module expressions. Its recursive calls are on structurally smaller expressions and base case are terms or blocks, for which typing is decidable. □

Chapter 9

Implementation in Coq

The practical aspect of the present thesis is the implementation of the module system in the Coq proof assistant. Modules are implemented in Coq version 7.4, officially released in February 2003.

The formalism presented in previous chapters has not yet been completely implemented. First of all the current implementation does not cover rewriting. Second, the module subtyping on signatures containing inductive types is less flexible in the implementation than in the theory, because inductive types cannot subsume types specified as `Parameter`. See discussion about correcting this deficiency in Section 12.2.

On the other hand the implementation covers several useful facilities, such as naming of module types, the notation `with` and most of all, interactive modules.

The extended abstract of this chapter has been accepted for presentation at the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003) [19].

9.1 The architecture of Coq

Coq is entirely written in Objective Caml [76]. The full system has about 120 000 lines of code organized in almost 500 files. The source tree is divided into the following directories:

kernel — the type-checker of the calculus of inductive constructions,

library — the backtracking mechanism, management of generic tables and objects, theory files and directories, and interface to the global environment,

parsing — abstract syntax trees and management of standard and user defined parsing and pretty-printing rules,

interp — translation from syntax-trees to untyped terms,

pretyping — analysis of untyped terms (generation of implicit arguments, etc.),

proofs — management of an interactive proof,

tactics — predefined tactics and the tactic language,

toplevel — interactive loop.

The kernel (about 10% of the code) is the only critical part of the system. Bugs in other parts of Coq cannot compromise the mathematical correctness of the theorem prover [42].

The implementation of modules follows closely this practice. Module type-checking has been added to the kernel, while the management of extra-logical module features is implemented outside, in the **library** directory.[1] Implementing the latter part turned out to be more difficult than implementing the type-checker. To cope with higher-order modules, a whole new mechanism of object substitutions has been introduced.

9.2 Modules in the kernel

In the type-theory tradition, the kernel of a proof assistant is the type-checker of the calculus implemented in the system. Up to version 7.3 the Coq kernel implemented the type-checker for the calculus of inductive constructions. With version 7.4, it also implements the type-checker for the module language.

The interface between the kernel and the rest of the proof assistant is the abstract type `safe_environment` representing correct environments, together with lookup functions and insert functions, which return an updated environment only if type-checking succeeds. The kernel is functional, i.e. no in-place modification is performed and the insert functions return the new environment or fail if type-checking is not successful.

Compared to the calculus presented in the previous chapters, the environment of the proof assistant contains more information than just names and specifications. In particular, in order to implement extraction,[2] implementations (proofs) must be present in the environment whenever given by the user.

An insert function, such as `add_constant`, `add_inductive` etc. (abbreviated by `add_*`), called with `add_*` E v S P, where E is a correct environment, v the name of the new element, S its specification and P an optional implementation[3], implements the following typing rule:

$$(\textsc{Add}) \qquad \frac{E \vdash \mathtt{ok} \qquad E \vdash S : \mathtt{spec} \qquad E \vdash P : S}{E; v : S := P \ \vdash\ \mathtt{ok}} \quad \text{if } v \notin E$$

It is important to note here that even though the environment sometimes contains implementations, they are never used for typing. Note also that when the implementation P is removed from the (Add) rule, we get the rule **(Env/Insert)** from our calculus.

Adding modules requires the function `add_module` to be implemented, together with the module type-checker. Like previous `add_*` functions, this can be seen as implementing the rule (Add), with P a module body, and S a module specification.

[1] Beware! There is a file called **library** in the directory **library**.

[2] In Coq it is possible to extract the executable content from a proof of validity of a program specification

[3] For the sake of clarity we ignore the fact that parts of the specification may be synthesized.

However, it would be difficult for the user to write the whole definition of a module at once, especially for large modules containing many definitions and proofs. Moreover, it would contradict the interactive character of Coq.

9.2.1 Interactive modules

What seems natural in an interactive theorem prover is *interactive modules*. The user should have a possibility to declare the beginning of a new module, input definitions, declare and interactively prove some lemmas and then close the module, making all the entered definitions and lemmas become the components of the newly defined module. This is exactly what is implemented in version 7.4 of Coq. The following Coq session (on the left) results in a module body m (on the right) assigned to the name `A`:

```
Coq < Module A.
Interactive Module A started
Coq <   Definition T:=nat.
T is defined
Coq <   Definition x:=O.
x is defined
Coq <   Definition y:bool.
1 subgoal

  ============================
   bool
Coq <     Exact true.
Subtree proved!
Coq <   Defined.
Exact true.
y is defined
Coq < End A.
Module A is defined
```

```
Struct
  Definition T:Set=nat.
  Definition x:nat=O.
  Definition y:bool:=true.
End
```

After closing a module, its components can be accessed using the dot notation:

```
Coq < Print A.x.
A.x = (O)
     : nat
```

A naive way to implement the `End A` operation would be to recover the whole body of the structure outside the kernel, restore the environment to be the one before `Module A` and call `add_module` with the reconstructed module body and environment.

Let us see what this would mean in terms of kernel operations. Let us call E the environment before the `Module A` command. First, during the interactive session, the definitions of `T`, `x`, and `y` would have to be checked by the kernel. After `End A` the non-critical part of code would reconstruct the module body m and the kernel function `add_module` would be called with the environment E, the name `A` and the module body m. Like all `add_*` functions, `add_module` would type-check m before returning the new environment, according to the (ADD) rule. Now let us concentrate on the right premise of this rule. In our case P is m and therefore the rule **(Sig/Struct)** must be used, all the premises of which have already been verified during the interactive phase of module construction.

The naive implementation of module ending would make the kernel do the same work twice. For this reason we decided to turn the beginning and ending of the module into kernel operations, which allowed to avoid redundancy.

So the interface of the kernel to the rest of the system has been extended with the functions `start_module`, marking the beginning of a module, and `end_module`, implementing the rule:[4]

$$\frac{E; v_1 : S_1 := P_1 \ldots v_n : S_n := P_n \vdash \mathtt{ok}}{E;\ V\ :\ \mathsf{Ty}(\ \mathsf{Sig}\ v_1 : S_1 \ldots v_n : S_n\ \mathsf{End}\ )\ :=\ \mathsf{Struct}\ v_1 : S_1 := P_1 \ldots v_n : S_n := P_n\ \mathsf{End}\ \vdash\ \mathtt{ok}}$$

where V is the module name. The premise of this rule implies the correctness of the signature and the admissibility of this rule follows from the fact that the only possible derivation of the premise is by a sequence of (ADD) rules. Their right premises imply

$$E\ \vdash\ \mathsf{Struct}\ v_1 : S_1 := P_1 \ldots v_n : S_n := P_n\ \mathsf{End}\ :\ \mathsf{Sig}\ v_1 : S_1 \ldots v_n : S_n\ \mathsf{End}$$

by an application of **(Sig/Struct)**. Using (ADD) applied to $E \vdash \mathtt{ok}$ and the above, we get the desired conclusion.

The same interactive mechanism can also be used to declare functors. Given a signature `SIG` (on the right[5]), one can define an interactive functor (on the left):

```
Coq < Module Two[X:SIG][Y:SIG].
Interactive Module Two started

Coq <   Definition T:=X.T * Y.T.
Coq <   Definition x:=(X.x, Y.x).

Coq < End Two.
Module Two is defined
```

```
SIG = Sig
        Parameter T:Set.
        Parameter x:T.
      End
```

In this case the environment operation performed by the function `end_module` shortcuts not only the **(Sig/Struct)** rule, but also (l applications of) the **(Funsig/Functor)** rule, leading to the admissible rule:

$$\frac{E;\ V_1 : \mathsf{Ty}(M_1) \ldots V_l : \mathsf{Ty}(M_l);\ v_1 : S_1 := P_1 \ldots v_n : S_n := P_n\ \vdash\ \mathtt{ok}}{\begin{array}{l} E;\ V\ \ : \mathsf{Ty}(\ \mathsf{Funsig}(V_1 : M_1) \ldots \mathsf{Funsig}(V_l : M_l) \\ \qquad\qquad \mathsf{Sig}\ v_1 : S_1 \ldots v_n : S_n\ \mathsf{End}\ ) \\ \quad := \mathsf{Functor}[V_1 : M_1] \ldots \mathsf{Functor}[V_l : M_l] \\ \qquad\quad \mathsf{Struct}\ v_1 : S_1 := P_1 \ldots v_n : S_n := P_n\ \mathsf{End}\ \ \vdash\ \ \mathtt{ok}\end{array}}$$

[4]In this section and the next one we do not distinguish labels and binders in structures and signatures.

[5]By the convention from Section 1.2, `Parameter v:`e is understood as a signature component $v : \mathsf{Ty}(e)$.

where $V_1, \ldots, V_l$ are formal parameters of the functor, $M_1, \ldots, M_l$ are their module types, and the triples $v_i : S_i := P_i$ are the functor body components.

An optional element of a module opening declaration is the expected module type. After the following definition:

```
Coq < Module A':SIG.
Coq <   Definition T:=bool.
Coq <   Definition x:=true.
Coq < End A'.
```

the signature of the module body is matched against the signature `SIG` and if the matching succeeds, the interface of `A'` is restricted to `SIG`, i.e. the specification given to this component is $\mathsf{Ty}(\mathtt{SIG})$. In our example it means that the conversion equalities `A'.T`$\approx$`bool` and `A'.x`$\approx$`true` are forgotten, making both these components abstract.

If a restricting signature is present, the implementation of the `end_module` function also involves application of the **(Subsumption)** rule.

9.2.2 Overview of the environment structure

In order to implement the function `end_module` the type representing correct environments has to contain the information about the number of definitions to be transformed into structure components, the number of module declarations to be turned into functor parameters and (optionally) against what signature the body must be matched.

Therefore the type `safe_environment` represents in fact a stack of growing correct environments $E_1 \subseteq \cdots \subseteq E_n$. Thanks to functional data structures and sharing, the elements of the environment E_1 are not copied in each $E_2, \ldots E_n$.

The operations `add_*` affect only the last environment, i.e. insertion in $E_1 \subseteq \cdots \subseteq E_{n-1} \subseteq E_n$ results in the environment stack $E_1 \subseteq \cdots \subseteq E_{n-1} \subseteq (E_n; v : S := P)$.

The operations which change the length of the environment stack are `start_module` and `end_module`. The first one is called with an environment stack $E_1 \subseteq \cdots \subseteq E_n$, a name for the new module V, its parameters $V_1 : M_1 \ldots V_l : M_l$ and optionally output signature M_{out}. The function verifies correctness of $M_1, \ldots M_l$ and M_{out} and the fact that names $V, V_1, \ldots, V_l$ are fresh and returns $E_1 \subseteq \cdots \subseteq E_n \subseteq E_{n+1}$, with

$$E_{n+1} = E_n; V_1 : \mathsf{Ty}(M_1) \ldots V_l : \mathsf{Ty}(M_l); \textsc{struct}\langle V, M_{out}\rangle$$

where $\textsc{struct}\langle V, M_{out}\rangle$ is a special mark that is transparent for typing rules. The mark is only used by `end_module` as a placeholder separating functor arguments from structure components and keeping the optional output signature M_{out}.

The correctness of the new environment follows from the fact that E_{n+1} can be obtained by applying l times the rule (Add).

The function `end_module` called with $E_1 \subseteq \cdots \subseteq E_{n-1} \subseteq E_n \subseteq E_{n+1}$, returns $E_1 \subseteq \cdots \subseteq E_{n-1} \subseteq E'_n$ where $E'_n = E_n; V : \mathsf{Ty}(M) := m$ with M and m synthesized from the information contained in E_{n+1}. More precisely, if E_{n+1} is

$$E_n; V_1 : \mathsf{Ty}(M_1) \ldots V_l : \mathsf{Ty}(M_l); \textsc{struct}\langle V, M_{out}\rangle; v_1 : S_1 := P_1 \ldots v_k : S_k := P_k$$

m and M become respectively

$$\mathsf{Functor}[V_1 : M_1] \ldots \mathsf{Functor}[V_l : M_l]\ \mathsf{Struct}\ v_1 : S_1 := P_1 \ldots v_k : S_k := P_k\ \mathsf{End}$$
$$\text{and} \qquad \mathsf{Funsig}(V_1 : M_1) \ldots \mathsf{Funsig}(V_l : M_l)\ M_{out}$$

after checking that $E_n; V_1 : \mathsf{Ty}(M_1) \ldots V_l : \mathsf{Ty}(M_l)\ \vdash\ \mathsf{Sig}\ v_1 : S_1 \ldots v_k : S_k\ \mathsf{End} <: M_{out}$. In case M_{out} is missing, M becomes

$$\mathsf{Funsig}(V_1 : M_1) \ldots \mathsf{Funsig}(V_l : M_l)\ \mathsf{Sig}\ v_1 : S_1 \ldots v_k : S_k\ \mathsf{End}$$

A similar mechanism is also used to implement interactive module types:

```
Coq < Module Type SIG.
Interactive Module Type SIG started
Coq <   Parameter T:Set.
T is assumed
Coq <   Parameter x:T.
x is assumed
Coq < End SIG.
Module Type SIG is defined
```

The function `start_modtype`, called with an environment stack $E_1 \subseteq \cdots \subseteq E_n$, a name for the module type V_M and the list of parameters $V_1 : M_1 \ldots V_l : M_l$ (nonempty, if one wants to define a functor type) verifies the correctness of module types $M_1, \ldots M_l$, the freshness of names $V_1, \ldots V_l$ and V_M and returns $E_1 \subseteq \cdots \subseteq E_n \subseteq E_{n+1}$, where $E_{n+1} = E_n; V_1 : M_1 \ldots V_l : M_l; \textsc{sig}\langle V_M \rangle$ where $\textsc{sig}\langle V_M \rangle$ is a mark as before.

The corresponding closing function, `end_modtype`, called with an environment stack $E_1 \subseteq \cdots \subseteq E_{n-1} \subseteq E_n \subseteq E_{n+1}$ returns the environment $E_1 \subseteq \cdots \subseteq E_{n-1} \subseteq E'_n$ where $E'_n = E_n; V_M \equiv M$ with M synthesized from E_{n+1}. More precisely, if E_{n+1} is

$$E_n; V_1 : \mathsf{Ty}(M_1) \ldots V_l : \mathsf{Ty}(M_l); \textsc{sig}\langle V_M \rangle; v_1 : S_1 \ldots v_k : S_k$$

M becomes

$$\mathsf{Funsig}(V_1 : M_1) \ldots \mathsf{Funsig}(V_l : M_l)\ \mathsf{Sig}\ v_1 : S_1 \ldots v_k : S_k\ \mathsf{End}$$

Since E_{n+1} is correct, we get $E_n \vdash M : \texttt{modtype}$ by applying **(Sig/Form)**, followed by l applications **(Funsig/Form)**.

Assigning names to module types is not covered by the theoretical part of this thesis. Unfolding of named module types may be treated like a textual macro expansion. It is implemented in the kernel only for efficiency reasons, i.e. the well-formedness of the abbreviated module types is checked only once.

Non-interactive module and module type definitions are implemented through the functions `add_module` and `add_modtype`. Note however that by the time of writing these words, parsing of whole structures and signatures is not implemented. Therefore non-interactive module definitions are basically limited to functor applications

```
Coq < Module DoubleA:=Two A A.
Module DoubleA is defined
```

and module renamings

```
Coq < Module A':=A.
Module A' is defined
```

The first one corresponds to an abstract module component DoubleA : $\mathsf{Ty}(M) :=$ `Two A A` where M is the module type inferred for the functor application using the **(Mod/App)** rule. The second definition corresponds to a manifest module component A' : $\mathsf{Eq}(\mathtt{A} : M_{\mathtt{A}}/\mathtt{A}) := \mathtt{A}$, where $M_{\mathtt{A}}$ is the principal module type of A.

9.2.3 Avoiding name clashes

The possibility of having nested modules forces us to take care of name clashes. This problem has already been discussed in Section 1.5.2. Since the solutions we adopted are slightly different in the theoretical and practical parts of our work, we present the problem again. Consider the following nested module:

```
Module F.
  Definition T:=nat.
  Definition x:T:=O.
  Module N.
    Definition T:=bool.
    Definition y:=x.
    Definition z:T:=true.
  End N.
  Definition z:=N.y.
End F.
```

The problem here is the two occurrences of the name T in the scope of the inner module. The simplest solution of rejecting every piece of code having two equal short names would be too restrictive, because the names of module components are important and the user cannot change them arbitrarily.[6] The second solution consisting in hiding completely the previous definition with the new one is also impossible, because the old definition may be accessed indirectly, as shown by our example (the type of y is the first T). Using the Coq tradition to internally represent names together with the section path (Top.F.T for the first T and Top.F.N.T for the second) does not work well with functors (if F was a functor) and module renamings, like `Module G:=F`.

The solution adopted in the theoretical part of this thesis follows the idea of Harper and Lillibridge [48], also adopted by Courant [32]. In this writing the primary signature of

[6] This strict policy is applied for functor parameters, since once a functor is defined, its parameters names are no longer important.

the module F would look as follows:

$$
\begin{array}{l}
\mathsf{Sig} \\
\quad \mathtt{T} \triangleright T : \mathsf{Eq}(nat : \star) \\
\quad \mathtt{x} \triangleright x : \mathsf{Eq}(0 : T) \\
\quad \mathtt{N} \triangleright N : \mathsf{Ty}(\ \mathsf{Sig} \quad \mathtt{T} \triangleright T' : \mathsf{Eq}(bool : \star) \quad \mathtt{y} \triangleright y : \mathsf{Eq}(x : T) \quad \mathtt{z} \triangleright z : \mathsf{Eq}(true : T') \quad \mathsf{End}\ ) \\
\quad \mathtt{z} \triangleright z : \mathsf{Eq}(N.\mathtt{y} : T) \\
\mathsf{End}
\end{array}
$$

To calculate the specification of $F.\mathtt{z}$ one has to use the substitution $\{T \mapsto F.\mathtt{T}, x \mapsto F.\mathtt{x}, N \mapsto F.\mathtt{N}\}$ on $\mathsf{Eq}(N.\mathtt{y} : T)$, obtaining

$$F.\mathtt{z} : \mathsf{Eq}(F.\mathtt{N}.\mathtt{y} : F.\mathtt{T})$$

Leroy in [59] and in the implementation of Ocaml attaches a unique number to every component name. The name (character string) plays the role of label and the name with the unique number (called identifier) plays the role of binder:

$$
\begin{array}{l}
\mathsf{Sig} \\
\quad T_1 : \mathsf{Eq}(nat : \star) \\
\quad x_2 : \mathsf{Eq}(0 : T_1) \\
\quad N_3 : \mathsf{Ty}(\ \mathsf{Sig} \quad T_4 : \mathsf{Eq}(bool : \star) \quad y_5 : \mathsf{Eq}(x_2 : T_1) \quad z_6 : \mathsf{Eq}(true : T_4) \quad \mathsf{End}\ ) \\
\quad z_7 : \mathsf{Eq}(N_3.y : T_1) \\
\mathsf{End}
\end{array}
$$

To calculate the specification of $F_0.z$, one takes the substitution $\{T_1 \mapsto F_0.T, x_2 \mapsto F_0.x, N_3 \mapsto F_0.N\}$ and applies it to the specification given by the module type, obtaining again

$$F_0.z : \mathsf{Eq}(F_0.N.y : F_0.T)$$

The solution we adopted consists in assigning a unique number not to every component name, but only to the leading Sig keyword. The signature of F is then represented internally as follows:

$$
\begin{array}{l}
\mathsf{Sig}\boxed{1} \\
\quad T : \mathsf{Eq}(nat : \star) \\
\quad x : \mathsf{Eq}(0 : \boxed{1}.T) \\
\quad N : \mathsf{Ty}(\ \mathsf{Sig}\boxed{2} \quad T : \mathsf{Eq}(bool : \star) \quad y : \mathsf{Eq}(\boxed{1}.x : \boxed{1}.T) \quad z : \mathsf{Eq}(true : \boxed{2}.T) \quad \mathsf{End}\ ) \\
\quad z : \mathsf{Eq}(\boxed{1}.N.y : \boxed{1}.T) \\
\mathsf{End}
\end{array}
$$

Now, to calculate the specification of $\boxed{0}.F.z$ one only has to use the short substitution $\{\boxed{1} \mapsto \boxed{0}.F\}$, obtaining

$$\boxed{0}.F.z : \mathsf{Eq}(\boxed{0}.F.N.y : \boxed{0}.F.T)$$

In the above, $\boxed{0}$ is a unique identifier attached to the outer block, as the unique identifiers are not only attached to every Sig, but to every Struct and marks $\textsc{struct}\langle\cdot\rangle$ and $\textsc{sig}\langle\cdot\rangle$ as well. Moreover we assume that the first element of the environment is the mark $\textsc{struct}\langle\cdot\rangle$ with the self identifier $\boxed{0}$. In other words the environment stack is never empty, and the unique identifier of the current structure or signature is stored in the corresponding mark.

Of course the user never sees these unique identifiers. The name table mechanism, described in Section 9.3.5, efficiently separates user names from internal representation. It is also interesting to note that our unique identifiers bear a lot of similarities to the *self* identifier used in object oriented programming.

9.2.4 Modules and sections

Before implementing modules, the section mechanism was the only available mechanism to group together related definitions inside one file. The main advantage of sections, however, is a mechanism of abstracting individual definitions. The following code

```
Coq < Section S.
Coq <   Variable A:Set.
Coq <   Definition x := [a:A]a.
Coq <   Definition y := x.
Coq < End S.
x is discharged.
y is discharged.
```

is equivalent to

```
Coq < Definition x := [A:Set; a:A]a.
x is defined
Coq < Definition y := [A:Set](x A).
y is defined
```

Of course parametric modules (functors) do not subsume the functionality of sections. The interactions between the two abstracting mechanisms are not straightforward. When a section is inside a module (or a module type), the components of the module (or the module type) generated by the section are exactly those which result from the closing of the section.

On the other hand the vision of modules inside a section is not very clear. Abstracting a whole module with respect to a variable is not covered by the theory. On the other hand abstracting individual components of a module raises the question of interaction of *to-be-abstracted* modules with the ones defined outside the section.

So, for the time being we resigned from implementing modules inside sections, leaving only sections inside modules.

9.2.5 The *with* notation

In order to make it easy to enter similar module types, the *with* notation has been implemented. For example if `SIG` denotes the signature

```
Sig
  Parameter a:Set
  Declare Module N:SIG'
End
```

the module type `SIG with Definition a:=nat` is equivalent to the signature

```
Sig
  Definition a:Set:=nat
  Declare Module N:SIG'
End
```

and the module type `SIG with Module N:=M` is equivalent to

```
Sig
  Parameter a:Set
  Declare Module N:=M
End
```

provided that the type of `M` is a subtype of `SIG'`.

Module signatures modified by `with Module` or `with Definition` are evaluated to plain signatures in the process of type-checking.

9.2.6 Implementation details

We discuss in detail all the new components added to the kernel in order to be able to type-check modules. We start by describing the elements of the most low-level kernel file, i.e. **names**.

Representation of names. Thanks to the simplifying assumption that every environment represents an open structure or an open signature, every name of term level object (constant, inductive etc.) can be represented as an element of some possibly open module. Moreover in order to account for the possibility of having open sections inside modules, the abstract type of names of objects is represented by triples:

```
type kernel_name = module_path * dir_path * label
```

where `label` is just a character string, `dir_path` is a list of names of nested open sections, and `module_path` is the following variant type

```
type module_path =
  | MPfile of dir_path
  | MPbound of mod_bound_id
  | MPself of mod_self_id
  | MPdot of module_path * label
```

representing a module read from a file, a bound module (i.e. functor parameter), an open module or a submodule respectively. We also assume that the `dir_path` part of any `kernel_name` is empty unless the `module_path` is `MPself _`.

Substitutions. As it was introduced in Section 9.2.3, the substitutions are necessary to calculate the specification of a module component or to calculate the signature of a functor application result.

The abstract type `substitution` is represented by a dictionary from the type of unique identifiers (representation of abstract types `mod_bound_id` and `mod_self_id`) to `module_path`, and equipped with the functions for creating empty and singleton substitutions and with functions extending substitutions.

In order to apply the substitution, for every substitutive type a suitable substitution function must be defined. The first substitutive type is of course `module_path`, for which the function

```
val subst_mp : substitution → module_path → module_path
```

is defined.

Substitution functions for other substitutive types are in fact mappings of this one. This is the case for `kernel_name`, whose substitution function is

```
val subst_kn : substitution → kernel_name → kernel_name
```

The last element regarding the substitutions in the file **names** is the function for sequentially composing substitutions:

```
val join : substitution → substitution → substitution
```

such that for all substituting functions `subst_*`, we have

$$\texttt{subst\_*}\ (\texttt{join}\ \theta_1\ \theta_2)\ t = \texttt{subst\_*}\ \theta_2\ (\texttt{subst\_*}\ \theta_1\ t)$$

Environment and safe environment. In previous version of Coq, the type of environment `env` was declared in the file **environ**, all typing functions operated on `env` and then the type of environment was reexported abstractly in the file **safe_typing** as the type `safe_environment`.

In our implementation, we do the same thing, but instead of simply reexporting the type `env`, we implement in the file **safe_typing** the environment with marks and functions `start_module` and `end_module`.

The implementation of the basic environment `env` has not changed much. The only modifications are the removal of information regarding modules read from disk together with management functions `import` and `export` (moved to **safe_typing**), and addition of dictionaries storing modules and module types.

The implementation of the type `save_environment`, representing environment stacks (see Section 9.2.2), is a nested record structure, where each record represents one open module or module type and contains its name, other information like the list of parameters or the optional output signature and the list of components of current structure or signature.

Apart from these elements it contains the `env` environment representing the environment without marks that is used by all typing functions. For efficiency reasons, `env` contains precomputed specifications of all module components in substituted form.

To understand what actions are performed by `start_*` and `end_*` functions, consider the following code fragment.

```
Module Type S.
  Definition T:=nat.
  Declare Module M.
    Parameter a:T.
    Definition b:=a.
  End M.
End S.
```

The (simplified) form of the safe environment before and after `End M` is shown in Figure 9.1.

The functions `start_*` add a new record to the list. The functions `end_*` remove the most recent record and add one declaration to the component list and `env` of the previous record. If the ended element was a structure, information about its components is added to the `env` as well.

Modifications in term typing. As the module language and the term language are independent (the module language is *on top* of the base language), the existing code for type-checking terms was barely changed.

The change in the representation of names did not force any modifications of the algorithm itself. The `kernel_names` are used in the term typing algorithm in an abstract way, just as a type with equality. This was the case in previous versions of Coq, where names were represented by the type `section_path`, and this is the case now.

From the theoretical point of view the type `kernel_name` corresponds to term access paths. The precomputation of substituted specifications for `kernel_names` follows the principal derivations of the specifications for access paths using rules **(P/Env/Lookup)**, **(P/Sig/Access)** and **(P/Ty/Type)**, **(P/Eq/Type)** for module specifications. Apart from precomputing the specifications, the term typing algorithm does not use module typing at all. The correctness of this approach is proved in Lemma 8.5.1. Indeed the algorithm works as indicated by the proof of this lemma: precomputation corresponds to signature flattening (Lemma 8.3.3) and the treatment of manifest modules follows Lemma 8.3.5. Even though the modules stay in the environment, the term typing algorithm works as if they were removed.

A more profound change in the typing algorithm was due to the notion of equivalence of inductive definitions. In the following situation

```
Module M.
  Inductive T : Set := a : T | b : T.
  Definition f := [x:T] Cases x of a => true | b => false end.
End M.

Module N:=M.
```

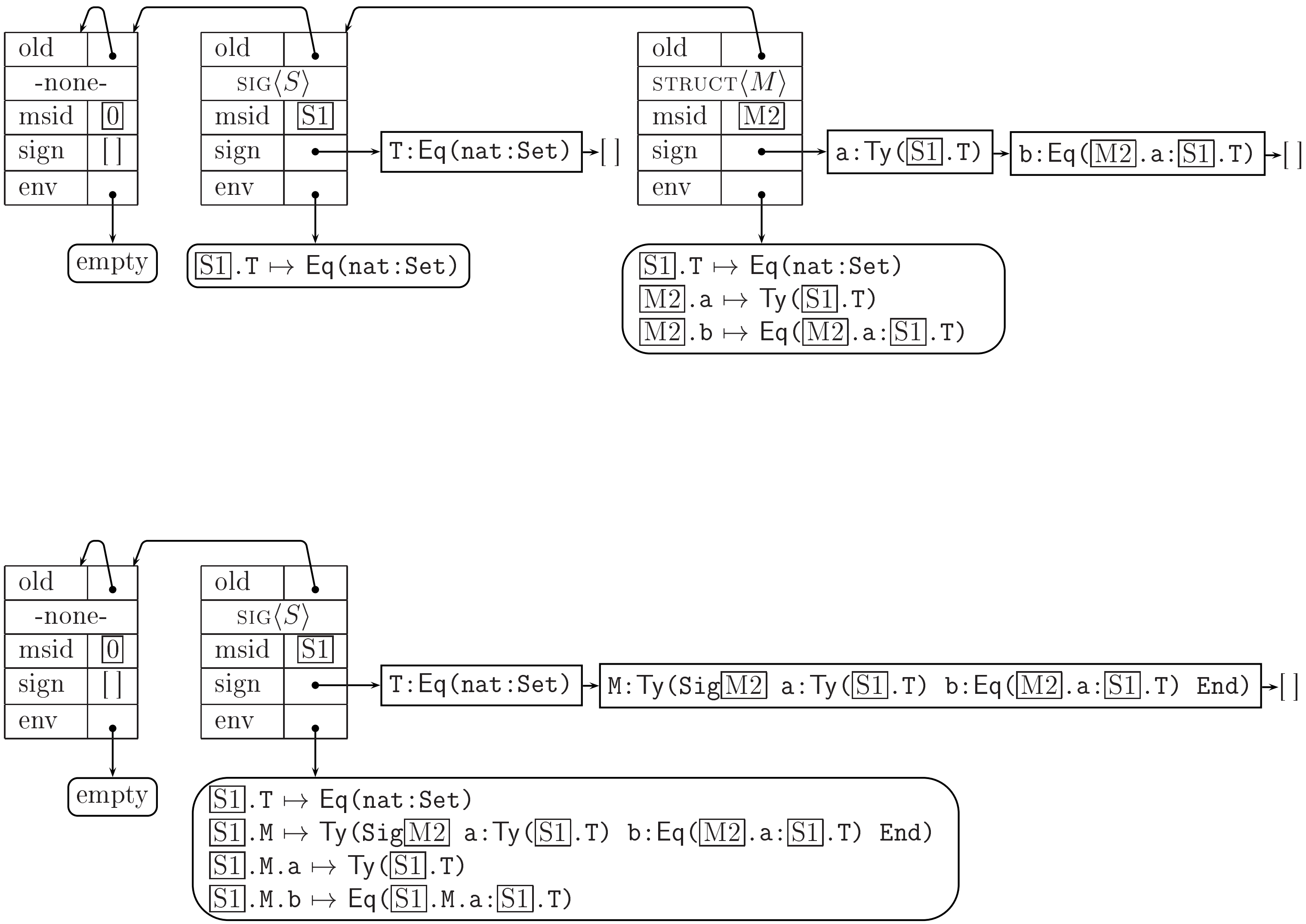

Figure 9.1: The environment before (left) and after (right) `End M`.

the term (`M.f N.a`) should be well-typed. To achieve this the inductive definition in the module `N` must be given a manifest specification, making `N.T` and `M.T` equal, together with their constructors.

The typing algorithm does not use the results of Section 8.4 about removing manifest block specifications. Instead, manifest inductive blocks are treated directly by the conversion algorithm.

This equivalence between inductive blocks is marked in a new field `mind_equiv` in the record type representing inductive definitions, `mutual_inductive_body`, in the file **declarations**. The convertibility test was changed accordingly, i.e. a new function `mind_equiv` was added in the file **closure** for testing the equivalence and the cases for inductive types and constructors were modified in the function `eqappr` in the file **reduction**.

Module typing. The topmost file of the module type-checking is **mod_typing**. It implements functions

```
val translate_modtype : env → module_type_entry → module_type_body
val translate_module : env → module_entry → module_body
```

which type-check the information given by the user (`*_entry`) and translate it to the form stored in the environment (`*_body`) or report an error if the given information is incorrect. These functions use the typing of the calculus of inductive constructions (in file **term_typing**), module **subtyping** and various auxiliary and error reporting functions from the file **modops**.

The type-checking algorithm is mostly standard. It closely follows the typing algorithm presented in [62] and implemented in Ocaml. The only difference is the careful treatment of the predicative universe hierarchy constraints.

Universe constraints. The current implementation of universe constraints is not completely satisfactory from modularity point of view. In particular the implementation of a module may introduce important universe constraints that are not reflected by the specification. Therefore the correctness of an application of a functor to a module is not guaranteed by the successful subtyping. One also has to verify that constraints generated by the functor and the module are not contradictory. Therefore modifications of a module which do not change the module interface do require retype-checking of all uses of the module, which contradicts modularity.

To guarantee the correctness of module type-checking with respect to universe constraints, we collect them at the module subtyping and store them in the structure representing translated module expressions.

All functions in the file **subtyping** take a set of constraints C as the first argument, some parameters needed for comparison (like the environment etc.) and then two elements e_1 and e_2 to compare. They return the set C augmented by constraints resulting from the comparison $e_1 <: e_2$. In this way the constraints are accumulated throughout the exported `check_subtypes` function.

Subtyping is essentially performed at two occasions: while type-checking a functor application and while defining a module with the explicit expected module type.

In the first situation the subtyping constraints are stored in the third component of the `MEBapply` constructor, representing the application node in the type `module_expr_body` in the file **declarations**. The latter type represents type-checked module expressions.

The constraints resulting from the second situation, where subtyping is used, are stored in the field `mod_constraints` of type `module_body` in the file **declarations**. This type represents the verified module definition, i.e. its specification and implementation.

In the end the constraints are duly added to the environment by `full_add_module` implemented in **safe_typing**. The constraints are those stored in the module expression, in the module type and resulting from subtyping. Functions `add_*_constraints` in the file **mod_typing** are responsible for the traversal of the module expression and the module type in quest for the constraints stored in application nodes of the module expressions and components of structures and signatures.

File modules — libraries. Modules stored on disk are treated very much like all other modules. To avoid misunderstanding they are called *libraries*. The only difference outside the file **safe_typing** is that unlike other modules, libraries are given the absolute module name `MPfile _`, and are not elements of a higher level structure.

In previous version of Coq, the code dealing with libraries was situated in file **environ**. In the current version all the code is in file `safe_typing`. In fact the code for loading libraries (in function `import`) is almost identical to adding any other module once we know it is type correct. The only difference about disk modules is that they have to be reloaded every time the open structure or signature where they were loaded is closed:

```
Module M.
  Definition x:=[a:nat]a.
  Module N.
    Read File.
    Definition y:=(x File.number).
  End N.
End M.
```

In the above code fragment, the body of `File` is entered in the environment three times: inside `N`, after closing `N` and after closing `M`. This is inefficient, but may easily be avoided by the user by reading all libraries before opening any interactive modules. The reloading operations are taken care of by the functions `end_module` and `end_modtype`, through suitable operations on the `loads` safe environment field.

Compilation of disk modules bears also many similarities with usual modules. In fact it is implemented almost exactly like an interactive module. We start compilation by calling the function `start_library`, which is a simplified version of the function `start_module` (currently a library cannot be a functor). When the compilation is finished, the data of the toplevel structure is gathered and returned as an abstract type `compiled_library` through the function `export`.

To assure the integrity of libraries the list of dependent libraries together with their checksums is stored with every compiled module. Similarly when a library is loaded the checksums of its dependencies are verified. For the sake of efficiency, however, the kernel does not calculate the checksums itself but relies on the caller of the `import` function. Of course, sacrificing efficiency, it can be changed to make the kernel even more self-contained.

It is important to notice that treatment of libraries like other modules enables further evolution of libraries from simple structures to possibly parametric structures (functors). It is also possible (putting universes apart) to split libraries into interfaces and implementations, like it is the case in Ocaml. There is absolutely nothing on the kernel side that prevents any of the above evolutions. The only modifications would have to be done in file **safe_typing**. Much more work would have to be done outside the kernel though.

Summary of kernel files. We will list here the more important changes in kernel files.

names — added definitions of types `module_path`, `kernel_name` and `substitution`; removed `section_path`.

term — added the substitution function `subst_mps`[7] on the type `constr`.

declarations — added new types representing correctly typed elements: `module_type_body` (module type expressions), `module_expr_body` (module expressions), `module_specification_body` (module specifications), `module_body` (module definitions), etc.;

added substitution functions on types `constant_body` and `mutual_inductive_body`;

`const_body` made lazy for efficiency of library loading;

added field `mind_equiv` for equivalence of inductive specifications.

environ — support for importing and exporting libraries moved to **safe_typing**; added dictionaries of modules and module types and functions dealing with them.

closure — added function `mind_equiv` for equivalence of inductive specifications.

reduction — modified two cases of `eqappr` to deal with equivalence of inductive specifications.

entries — new file: types representing non-type-checked inductive and constant definitions (`mutual_inductive_entry` and `constant_entry`) moved from **indtypes** and **safe_typing** respectively;

added types representing non-type-checked module related elements: `module_type_entry` (for module type expressions), `module_expr` (module expressions), `module_entry` (definition/declaration of a module).

[7] There are quite a few substitution functions for `constr`, dealing with de Bruijn indices, the section mechanism etc., so the name `subst_constr` would be too meaningless.

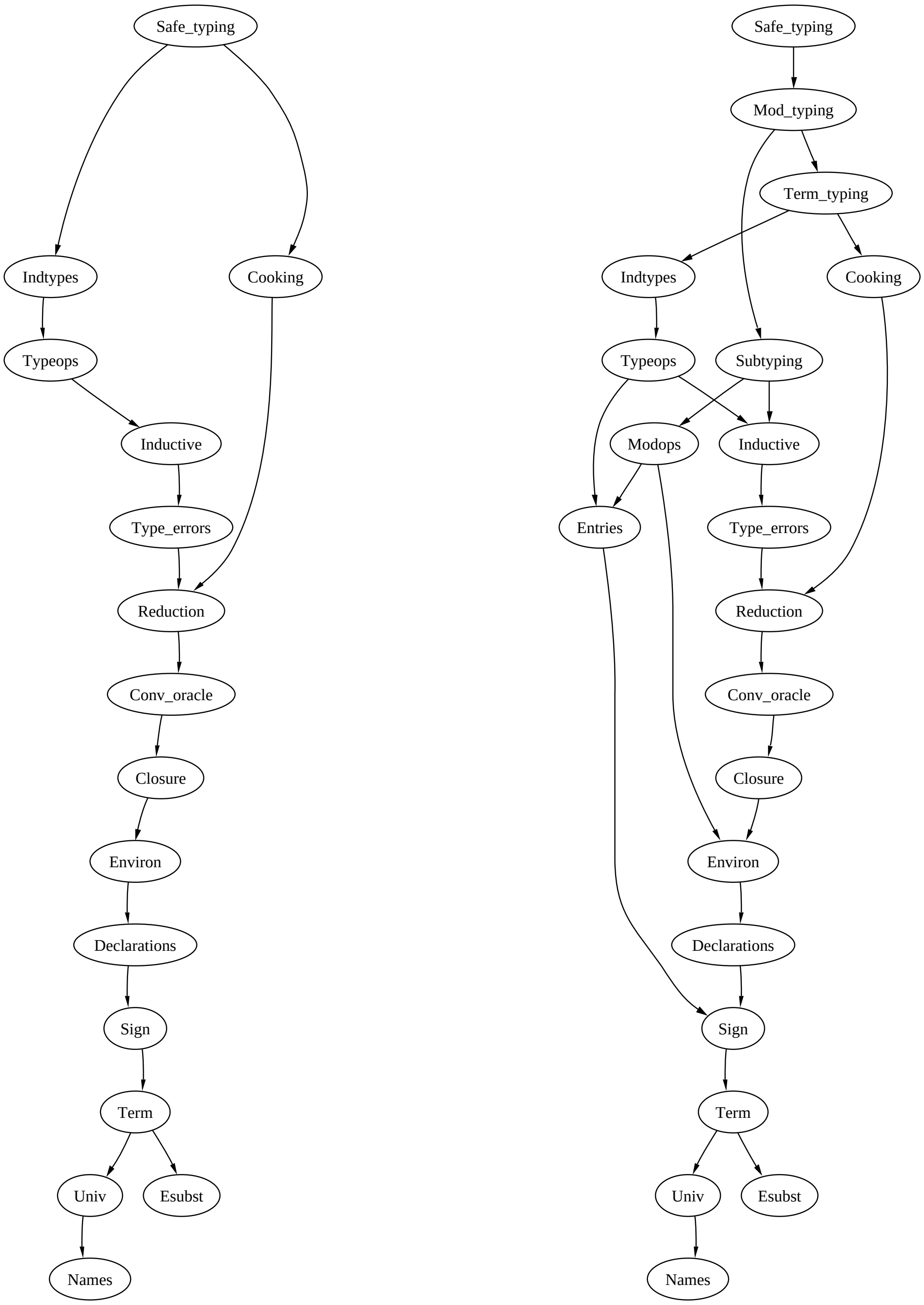

Figure 9.2: The old and the new structure of the kernel

modops — new file: contains functions for module and module type substitutions, for recursively adding module components to the environment, strengthening, signaling module errors and other auxiliary functions.

subtyping — new file: contains a function calculating universe constraints resulting from subtyping or calling an error function if subtyping incorrect.

term_typing — new file: contains functions translating constant and inductive entries to bodies and checking local assumptions and definitions (all of them moved from **safe_typing**).

mod_typing — new file: contains functions for translating module and module type entries to bodies and functions adding module and module type universe constraints to the environment.

safe_typing — added new functionality of the safe environment: addition of modules and module types, interactive modules and module types, importing and exporting libraries (moved from **environ**); some of the type-checking code moved to **term_typing**.

9.3 Modules outside the kernel

The most important challenge of the introduction of modules in Coq was not the implementation of the type-checking algorithm. Apart from managing the certified environment, Coq has extra-logical features like implicit arguments, an extensible pretty-printer and an extensible parser, databases for automated tactics, etc. They are controlled through commands which can very well be issued inside modules or module types. The management of these features in the presence of modules was technically much more difficult than the modifications in the kernel.

9.3.1 Overview

The generic mechanisms responsible for the management of extra-logical features of Coq are implemented in the directory **library**. The functionality of the files in this directory are the following:

1. Global environment

   The file **global** contains the reference to a safe environment and reexports the interface of the kernel file **safe_typing**, but after every operation, the value of the reference is set to the modified environment.

2. Backtracking and state management

   Backtracking is implemented through the generic extensible mechanisms of *library objects* and *summaries*.

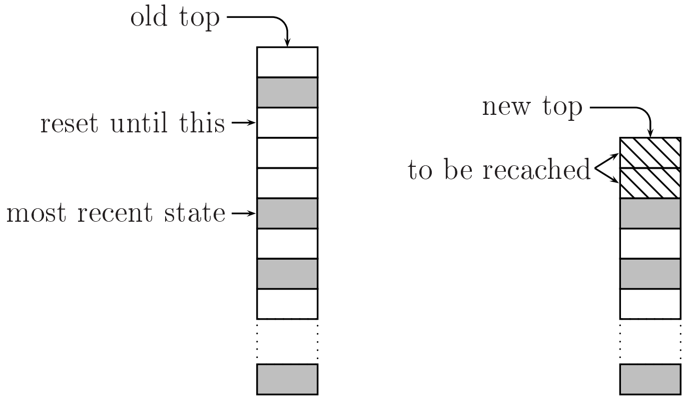

Figure 9.3: Backtracking

3. Theory file management

   Compiling theory files and loading them, management of dependencies and logical/physical paths. This is implemented in the file **library**.

4. Name visibility management

   Every Coq element has a unique long name, like `Coq.Datatypes.nat` for the type of natural numbers. The validity of its suffixes is stored in the name table, implemented in the file **nametab**.

5. Declaration of constants, variables etc.

6. Features: implicit arguments, generic options, etc.

The module operations on extra-logical features are implemented in a new file **declaremods**. Of course many other files were also modified. Before going into details, let us briefly present the generic mechanisms of Coq.

Backtracking and library objects.

Since Coq is an interactive proof assistant, it is important to let users correct the mistakes they make. Therefore a backtracking mechanism is implemented in Coq (through the `Reset` command).

The commands issued by the user are pushed on a stack, and from time to time, the full state of the system is pushed there as well. To backtrack to a given point it is enough to "recall" the most recent copy of the state preceding the point and redo the commands following the state and preceding the point. See Figure 9.3 for a graphical presentation (copies of the state are marked gray, and regular commands – white).

In order to make this mechanism work properly and be easily extensible, the "commands on the stack" are given a common interface of so called *library objects*[8] implemented in the

[8]In fact for historical reasons, they are not Objective Caml objects.

file **libobject**. The object method `cache_function` implements the non-functional part of command execution (usually table updates) and is called when pushing the object to the command stack (by the function `add_leaf`), and while backtracking (by the function `reset` – both functions in the file **lib**.

In previous versions of Coq the only other uses of the generic object mechanism was to store and restore (to and from the disk) the extra-logical elements of compiled theory files (methods `load_function` and `open_function`) and to partially control the discharge at the end of a section (the method `export_function`).

Summaries and state

An important aspect of the implementation of backtracking is the possibility to periodically store the state of the system. Various Coq features require mutable information to work properly, for example the `Auto` tactic uses hint databases, the pretty-printing mechanism uses tables of pretty-printing rules, parsing — of parsing rules etc. Appropriate commands change the contents of these tables, so the contents of each table is a part of the state, and therefore should be stored. Of course copying the contents of each table would waste a lot of memory, so the data-structures used in Coq are all functional and their "copying" is just a pointer copying.

In order to facilitate the addition of new features in Coq, the tables able to store and restore their contents are given a common interface of so called *summaries*, implemented in the file **summary**. The interface specifies three methods `freeze_function`, `unfreeze_function` and `init_function` for storing, restoring and initializing a table respectively, and one boolean value `survive_section` which partially controls the discharge at the end of a section. The important remark here is that the global environment is also registered as a summary and managed through the generic mechanism.

9.3.2 Module operations on objects

To understand the issues we had to solve when putting together modules and extra-logical features of Coq, consider the following code which uses `Syntactic Definition` as an example.

```
Coq < Module M.
Coq <   Definition id := [A:Set][a:A]a.
Coq <   Syntactic Definition Id := (id ?).
Coq <   Check (Id O).
(id nat (O))
     : nat
Coq < End M.
Coq < Check (M.Id true).
(M.id bool true)
     : bool
Coq < Module N:=M.
```

```
Coq < Check (N.Id (M.Id None)).
(N.id (A:Set)(option A) (M.id (A:Set)(option A) None))
     : (A:Set)(option A)
Coq < Print N.Id.
 Syntactic Definition N.Id = (N.id ?)
```

When we rebind the module `M` to a new name `N`, in the same time we create a new syntactic definition `N.Id`, which now is a shorthand for `(N.id ?)`. So the same mechanism of substitutions that exists in the kernel for module components had to be implemented for library objects[9]. This mechanism naturally covers objects defined inside a functor body and used after the functor is applied to a module.

```
Coq < Module Type SIG.
Coq <   Parameter id : (A:Set)A->A.
Coq < End SIG.
Coq < Module F[X:SIG].
Coq <   Syntactic Definition FId := (X.id ?).
Coq < End F.
Coq < Module M1:=(F M).
Coq < Module N1:=(F N).

Coq < Check (M1.FId O).
(M.id nat (0))
     : nat
Coq < Check (N1.FId true).
(N.id bool true)
     : bool
```

Moreover, it is possible to attach objects to a module type. Such objects can be used inside a functor body or attached to a module defined with that module type.

```
Coq < Module Type OBJECTS.
Coq <   Parameter id : (A:Set)A->A.
Coq <   Syntactic Definition Id := (id ?).
Coq < End OBJECTS.
Coq < Module G[X:OBJECTS].
Coq <   Definition id := [A:Set][a:A](X.Id a).
Coq < End G.
Coq < Module X.
Coq <   Definition id := [A:Set][a:A]a.
Coq < End X.
Coq < Module P : OBJECTS := (G X).

Coq < Check (P.Id O).
(P.id nat (0))
     : nat
```

[9]Moreover, the same substitution that is used in the kernel, is also used in the library.

The interface of objects has been extended with a new method `subst_function`, and in order to unify the treatment of objects for simple modules, functors and module types, we introduced a concept of *object closure*. It corresponds to a list of inactive objects that reference module names invalid in the current context, that wait to be substituted once the objects are assigned to an actual module and activated. An object closure is remembered for simple modules because of renamings, for functors because of applications, and for module types because of their possible attachment to modules.

Technically, an object closure is built from a list of objects, a list of bound module identifiers (functor formal parameters), a *self* identifier and a delayed substitution for efficient handling of partially substituted object closures.

In the rest of this chapter we call objects which behave like our `Syntactic Definition` *substitutive*. However, we do not want to force all kinds of objects to have this behavior. Therefore we decided to leave the object writers the choice: either the object must be equipped with a substitution function or it may only refer to "user names" that do not contain module identifiers (see Section 9.2.3). The objects of the latter kind are dropped during the operations which require renamings and therefore are useless in functor bodies and module types.

9.3.3 Interactive modules revisited

When the user starts a new interactive module, `Declaremods.start_module` is called, which itself calls `Global.start_module` to start the module in the global environment and `Lib.start_module` to push a special start mark on top of the library stack, together with the frozen state.

Inside the module, user commands are executed and corresponding objects are pushed on the stack and their `cache_function` methods are called.

When the module is ended, `Declaremods.end_module` is called. Its operations are the following.

1. End the module in the global environment,
2. Get objects and the state from the library stack,
3. Classify objects,
4. Compute the object closure,
5. Prepare the module object,
6. Restore summaries except for the environment,
7. Push the module object on the stack.

The operation on the global environment is performed through `Global.end_module` (point 1). The call to `Lib.end_module` recovers all objects[10] pushed on the stack after the most recent module has been started and the state of the system stored at that time (point 2).

The objects are divided into four groups according to their classification (by the function `Lib.classify_segment`), one of them is disposed and three of them are kept (point 3). These are *substitutive* objects, *non-substitutive* objects and *anticipated* objects. The latter group are in fact library objects associated with the commands `Require` and `Read`.

Next, the object closure of the module is computed (point 4), according to the module specification: If the module was given an output signature when it was started, we take the object closure of the signature. Otherwise the object closure is constructed from the objects recovered in point 2, the list of functor arguments given when the module was started, the *self* identifier of the module body and the empty substitution.

Now, if the ended module is not a functor, the objects from its object closure are substituted (using the method `subst_function`) with the substitution contained within the closure extended with the mapping of the *self* identifier to the name of the module. The object closure, the substituted objects and the non-substitutive objects are packaged into a module object (point 5), but before putting it on the command stack the summary operation is performed.

Through the new summary function `unfreeze_other_summaries`, all tables except for the global environment are restored (point 6) to the state recovered in point 2.

Only now, the module object is pushed on the stack, just after the anticipated objects and their `load_function` methods are called (point 7). For the module object, this method first of all stores the object closure in the table `modtab_substobjs` and if the module was not a functor, calls `load_function` for every substituted and non-substitutive object. This way if the module has submodules, their objects are activated as well. Both lists of substituted and non-substitutive objects are stored in the table `modtab_objects` which is kept for a possible future `Import` operation (see below) or other operations requiring searching through objects attached to a module.

9.3.4 Non-interactive modules

Although the operations described in the previous section may seem complicated, our goal was to maximally unify the object operations for interactive and non-interactive modules (possibly nested), for modules loaded from disk, and for module types, having in mind that all these should cooperate with the backtracking mechanism.

Indeed, the management of non-interactive modules is similar to the points 4, 5 and 7 of the `end_module` operation. Let us take

```
Module M:=F N
```

[10]Intermediate copies of the state stored on the stack inside the module are forgotten.

as an example, and suppose that the formal parameter of the functor F is X. The object closure of M is constructed from the object closure of F, by extending the substitution contained within the closure with the mapping from X to N[11]. Now the module object is prepared in a similar way to the point 5 in the previous section, with the only exception that there are no non-substitutive objects and that the module object contains also the body of the module definition, that will be passed to the kernel.

When the module object is pushed to a stack, the method cache_function of the module object is called. It starts by updating the global environment with the function Global.add_module and then performs exactly the same operations as the load_function called in the end of the end_module operation: the object closure is stored in the table modtab_substobjs, the substituted objects in modtab_objects, and load_function is called for every one of them.

Theory files are treated like special modules and their object operations are performed by module functions. Interactive and non-interactive module type operations are very similar to those for modules, but simpler, since only the object closure is important and no objects are substituted. For module types there is only one table, modtypetab storing object closures. Also, non-substitutive objects resulting from commands issued inside an interactive module type are simply discarded.

The last module operation affecting library objects is Import. This command declares all components of the module to be directly available at toplevel without any prefix. This command was already available in previous versions of Coq, but concerned theory files only. Now it has been naturally extended to all modules.

Importing a module may also affect activity of some objects even though they have no name available to the user. For example it is natural to expect that parsing or pretty printing rules defined inside a module should be active only if the module is imported.

```
Coq < Module P.
Coq <   Definition two:=(S (S O)).
Coq <   Notation "# x":=(plus x two).
Coq < End P.

Coq < Eval Compute in (#two).
Toplevel input, characters 17-18
> Eval Compute in (#two).
>                  ^
Syntax error: [test_int_rparen] or [constr:lconstr] expected after '(' (in [constr
:constr])
Coq < Import P.
Coq < Eval Compute in (#two).
     = (4)
     : nat
```

[11]If we had a module renaming, Module M:=N, we would take the object closure of N, and if the output type was given, Module M:SIG:=..., we would take the object closure of SIG.

For that purpose, every object declares the method open_function, which implements the action to be performed when the module containing the object is imported. The function import_module realizing the command Import, looks up the module objects in modtab_objects and calls open_function on each of them.

9.3.5 Long names and suffix visibility

Since version 7 of Coq, all named elements, like definitions, inductive types etc. have unique long names available to the user. For example the long name of the inductive type list from the file **PolyList** of the standard library is Coq.Lists.PolyList.list. Depending on the way the file **PolyList** gets loaded in the Coq toplevel, longer or shorter suffixes of this unique long name are also valid qualified identifiers referring to list. After the command Require PolyList all suffixes are valid, and after Read Module PolyList the shortest valid suffix is PolyList.list. Note that in the latter case the command Import PolyList makes the shortest suffix, list, valid as well.

Without modules, suffix validity patterns were very simple: either all suffixes were valid, or all suffixes of length strictly greater than one. In the recent version of Coq, visibility patterns may be more complicated.

After entering the following example (on the left), the validity of suffixes of Top.M.N.P.z becomes (on the right):

```
Module M.
  Module N.
    Module P.
      Definition z:=O.
    End P.
  End N.
End M.

Import M.N.
```

Top.M.N.P.z	valid
M.N.P.z	valid
N.P.z	invalid
P.z	valid
z	invalid

Of course suitable Import sequences may lead to arbitrary validity patterns with the only exception that all suffixes containing the leftmost occurrence of a module identifier are valid. In our example M is the leftmost module name, and Top is the logical path, and in the first example of this section, PolyList is the module identifier, while Coq.Lists is the logical path. At the same time, notice that module operations may request the validity of either all suffixes containing the given minimal suffix or of precisely the given suffix.

Validity of suffixes is managed by the file **nametab**, whose function is to maintain the correspondence between user names and internal names. During the implementation of modules, the file **nametab** has been substantially cleaned up and its functionality extended.

Fully qualified long names, like Top.M.N.P.z, are represented by one of the two types: Libnames.section_path for simple elements and Libnames.dir_path for modules. The abstract absolute kernel names are represented by the types Names.kernel_name and Name.module_path respectively.

In general, **nametab** implements registration functions and two kinds of translations: from partially qualified user names to kernel names and from kernel names to user names. The latter translation comes in two flavors: the result is either the full user name of the object or the shortest suffix referring to the same object.

All registration functions have a parameter of type `visibility`, which corresponds to two kinds of suffix validity, possibly requested by module operations.

```
type visibility = Until of int | Exactly of int
```

where the information about the given suffix is its length. The length of the corresponding suffix is passed to object methods `load_function` and `open_function` which register the objects name with visibility `Until` and `Exactly` respectively.

In the first example of this section, the command `Read Module PolyList` results in a call to `load_function` of the object representing the inductive type `list` and the latter registers the name with the visibility `Until 2`. The consecutive `Import` results in a call to `open_function` and registration with `Exactly 1`. In the second example, closing of the module `M` registers `z` with `Until 4`, and `Import M.N` with `Exactly 2`.

9.3.6 Summary of library files

Here is the list of more important changes made to files in library due to module introduction

nameops — all definitions moved to libnames; now contains mostly operations on identifiers

libnames — new file: contains definitions of qualified user names moved from the file **nameops** and the kernel file **names**

summary — added special treatment for the global environment

nametab — added tables for modules, module types and directories; generally cleaned-up

libobject — added two new methods in object interface: functions `classify_function` and `substitute_function`

global — added new functions reflecting changes done in the kernel file `safe_typing`

lib — added functionality connected to interactive modules: new stack nodes, functions for starting and ending modules and module types

declaremods — new file: contains all functions managing extra-logical aspects of modules

library — most operations on the state are now performed through **declaremods**

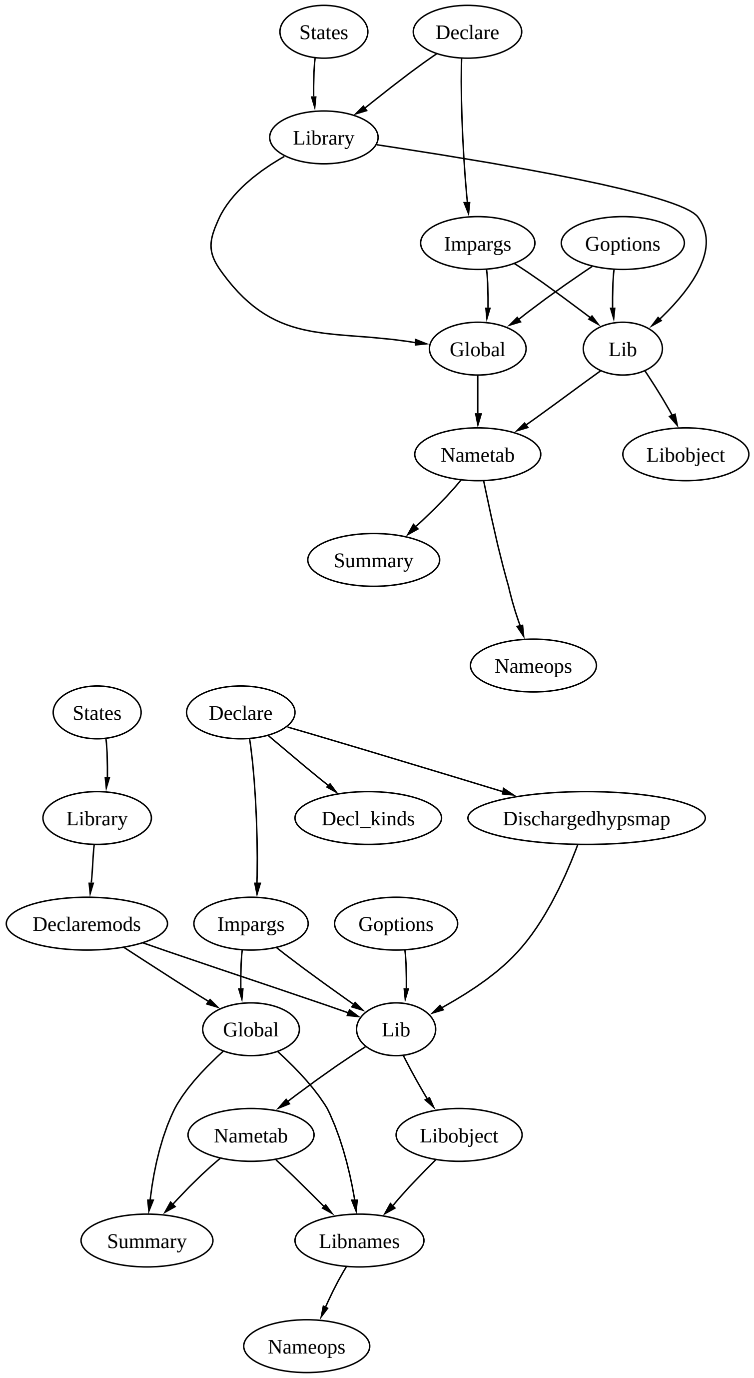

Figure 9.4: Old (top) and new (bottom) structure of the library

9.4 Conclusions

In this chapter we have described the implementation of our module system in the Coq proof assistant. The implementation carefully follows the idea of separating correctness-critical code (the kernel) from the rest of the system.

Thanks to interactive modules the user can construct parametric theories and certified data structures without losing the advantage of interactive proof development.

The modules have been successfully used in several examples, even though some features, like parsing and printing of structures and signatures and precise error messages are still not implemented.

On the other hand, generic Coq mechanisms of objects and summaries has been adapted to the new modular structure and most user facilities already cooperate with modules in a satisfactory way.

Chapter 10

Related work

10.1 Module systems for ML

The module system studied in this thesis, which is an adaptation of $\mathcal{MC}$ from [32], is a variant of module systems from the ML family of programming languages.

For the first time modules were introduced in the Standard ML [70] of New Jersey [2] in the mid-eighties [65] and the semantics of modules as dependent records was given in [66]. Since then the module language evolved gradually through the introduction of higher order functors [67] and type abbreviations in signatures [59, 48]. In parallel, the notion of type abstraction was studied in details [72, 17, 16] and modules became the main vehicle of type abstraction in ML [61]. The overly complicated stamp-based semantics of functor application has been simplified using type-theoretical concepts: translucent sums [48, 63] and manifest types [59].

Next, the traditional generative semantics of functors was questionned: applicative functors have been implemented in Objective Caml [60], and in Moscow ML [73] a programmer can choose for each functor whether it will be generative or applicative [89]. Moscow ML implements also modules as first-class objects [89, 90], a feature which has already been studied in [48, 63] and earlier in [71].

Recently, the problem of cross-module recursion is getting more and more interest [36, 40]. One approach consists in using the so-called mixin modules [1, 50], which are partially defined structures with the operation of "mixing" two partial structures, replacing fields undefined in one structure with the definitions from the other and vice versa. Another, more direct approach has been used to implement recursive modules in Moscow ML [91] and Ocaml [51].

The newest features of ML-style modules do not appear to be much wanted for inclusion in proof assistants. The need for these features will probably arise when formal proofs of correctness of programs requiring cross-module recursion will start to be developed using proof assistants. For now, most proof assistants provide modular features comparable to signatures/structures with first order functors.

10.2 Theory management in proof assistants

10.2.1 A module system for ELF

The LF Logical Framework [47] is designed to provide an appropriate language for the high-level specification of deductive systems. The framework is intentionally weak: there is no polymorphism, impredicative constructs etc. This permitted the construction of operational interpretation of LF realized in a programming language Elf [83, 84].

The module system for Elf [49] is built from two principal elements: signatures and realizors, which correspond to first order functors.

Signatures define axioms and rules corresponding to modelled deduction systems. Realizors provide a means of interpreting one signature into another. For each declaration in the target signature, a definition must be provided using the constants from the source signature. Signatures and realizors can be parametrized and instantiated by other realizors.

Signatures cannot have manifest values but these can be expressed by parametrization. On the other hand, the language contains operations to include one signature in the other, making signature inclusion explicit. This feature is usually absent in theoretical descriptions of the ML module language.[1]

10.2.2 PVS theories

Prototype Verification System PVS [78] is probably the most widely used of all proof assistants. It contains a very expressive specification language based on dependently typed higher order logic with predicate subtyping. Its type-checking is undecidable and hence verifying well-formedness of statements may generate type-checking conditions called TCCs, which have to be proved by the user.

Theories are a very important concept in PVS. Their semantics have been formally defined in [79]. Every development consists of one or more theories. A theory contains three main parts: parameters, assumptions and exports. When the theory is imported its parameters must be explicitely instanciated and assumptions generate TCCs. In return, the definitions and lemmas from exports can be used freely.

Speaking in the module language, a PVS theory may be interpreted as a first order functor, whose input signature consists of parameters and assumptions of the theory and output structure is the exports part. However, assumptions and exports may include other theories, which corresponds to inclusion of signatures and structures.

10.2.3 Maude's module algebra

The module system implemented in Maude [69] is described in [41]. Its semantics is presented using the powerful categorical notion of institutions [46].

Its main ingredients are modules, theories and views. A module is a specification with initial or free algebra model. A theory is a specification with a "loose" model, i.e. any

[1] However, signature and structure inclusion is implemented in Ocaml.

algebra verifying the specification is a model of the theory. Views are theory interpretations used to instantiate parameter theories, refine specifications and assert formal properties.

In the theoretical approach all three concepts can be parametrized and instantiated by theories, but implementations provide only instantiation of modules.

We can say that Maude modules roughly correspond to structures and functors from the ML module language (they are precisely defined), theories to signatures and parametric signatures (they specify a class of structures) and views, being algebraic signature morphisms, correspond to coercions with the possibility of renaming. They replace signature subtyping from the ML module language.

10.2.4 Isabelle

The main concept for structuring mathematical developments in Isabelle [52] is a theory. Theories organize the syntax, declarations and axioms of the mathematical development. They are built by extending and merging existing theories (starting from one or two built-in ones).

Isabelle provides two more means to structure mathematical developments: locales and axiomatic type classes. Locales [56] are mostly concerned with convenient management of assumptions. They correspond to Coq sections with extended functionnality. It is possible to reopen an already closed locale, have several open locales at the same time or interactively decide to abstract a given theorem with respect to the local assumptions. A proof of Sylov theorem [57], depending on many layers of algebraic structures, showed the usefulness of the new concept.

Another construction is that of axiomatic type classes [75]. They resemble Haskell type classes and provides very lightweight notation for operations and properties of the given type and its derivatives (like product, functional type etc.) This functionnality is strictly connected with a particular type, so it is for example not possible to consider at the same time the additive and the multiplicative monoids issued from a given ring (both declared as a monoid type class). From the theoretical point of view much of the functionnality of type classes can be simulated by first order functors, but practically such a simulation would mean to write a functor for each class and write a module for each instance of that class, apply the functor to the instance etc.

10.2.5 Dependent records

Dependent records have been proposed as a means to represent mathematical structures in [10, 9, 8]. Unlike in all previous approaches, structures are first class anonymous values, so there is no need to consider a particular notion of parametrization. This first proposal has been extended with manifest fields [86] and showed to be consistent by encoding in the type theory of inductive-recursive definitions.

Recently, dependent records with manifest fields have been presented in a logical framework based on the Martin-Löf type theory [30]. The semantics of the records is built using a PER model and the correctness of subtyping is proved using the fact that subtyping is a general form of η-expansion.

Dependent records have been implemented in Agda [25] and are used in a similar way to SML modules.

10.2.6 Discussion

To summarise the presented examples, we can say that many proof assistants provide a functionality similar to modules and functors with various syntactic facilities. Often inclusion of signatures and structures is provided which is currently missing in Coq.

As far as rewriting is concerned, the difficulty of adding it mostly depends on the nature of a tool, the logic, the treatment of equality rather than the structuring feature. In Maude rewriting is an inherent part of the logic and the implemented system of named modules has no interactions with it. In PVS and Isabelle the correctness of the system does not depend on the termination and confluence of reductions. At worst, a tactic invocation will loop and will have to be interrupted by the user.

From the structuring facilities decribed above, only Isabelle locales and dependent records are different from our modules in important aspects. Problems related with locales and rewriting are exactly the same as with Coq sections and rewriting. The easier part concerns abstracting definitions by rewriting with respect to local variables and assumptions and the harder part concerns allowing local definitions by rewriting and abstracting constants and theorems with respect to those definitions, which seems out of reach for current normalisation and confluence techniques.

Adding definitions by rewriting to dependent records would have the same negative effect as adding them to second-class anonymous modules (see Section 1.6 for discussion).

Chapter 11

Satisfying closure properties

In this chapter we discuss side-conditions for formation rules for inductive types and rewriting and closure properties they have to satisfy. This concerns typing rules **(Ind-Spec/Form)** with the side-condition POS, **(RewSpec/Form)** with the side-condition ACC and **(RewSpec/Rew)** with the side-condition COMP.

The role of the first two conditions is to assure that type-checking in a correct environment is decidable, and an environment is correct only if all definitions by rewriting have been checked by the condition ACC and all inductive definitions have been checked by the condition POS.

The third condition is supposed to assure that any formal development is consistent. By a formal development we mean a structure well-typed in the empty environment (see Definition 3.1.2). A structure is well-typed only if all definitions by rewriting, using Rew as an implementation, have been checked by the condition COMP (in addition to ACC).

In particular, this condition is supposed to guarantee, that the proposition t in the resulting structure in Theorem 1 in Chapter 3 is not `False`.

Our work shows, that given such conditions satisfying five closure properties defined in Section 2.7.1, if they indeed imply decidability of typing in an environment without modules,[1] then typing terms in any environment (with modules) is decidable.

Similarly, if the condition ACC guarantees consistency of a formal development without modules[2] then it also guarantees consistency of any formal development (with modules).

Below we discuss the closure properties that have to be satisfied by the acceptance conditions. Before going into the details it is important to remark that judgments in our calculus are also closed by operations defined in closure properties, i.e. $E \vdash J$ implies $E' \vdash J'$ where E and E' are as in the closure properties and the relation between J and J' is the same as between Ω and Ω'.

This means in practice that if an acceptance criterion requires performing some additional typing or convertibility tests on terms, they do not affect satisfaction of closure properties. Indeed, all judgments in our calculus are closed by substitution of paths to vari-

[1] In fact it is enough to show decidability of typing in a classic environment, i.e. without modules and without manifest block specifications.

[2] idem for consistency.

ables (Lemma 4.1.5) and by environment extension (Lemma 4.1.4) and all term judgments are closed by module removing operations (Lemmas 8.3.3, 8.3.4 and 8.3.5).

In the rest of this chapter we assume that a rewrite reduction $\longrightarrow_R$ and delta reduction $\longrightarrow_\delta$ are defined by orienting judgments obtained from the rule **(P/RewSpec/Comp)** and **(P/Eq/Comp)** respectively:

$$\frac{E \vdash_P \overline{p} : \mathsf{RewSpec}(E^\Gamma, R) \qquad \theta : E \to E_i}{E \vdash e_i^L\{v_j \mapsto p^j\}\theta \longrightarrow_R e_i^R\{v_j \mapsto p^j\}\theta}$$

$$\frac{E \vdash_P \overline{p} : \mathsf{Eq}(\overline{\phi} : \Phi)}{E \vdash p^i \longrightarrow_\delta \phi^i}$$

It is important to remark that reductions defined as above are closed by module removal operations, because judgments in their premises are.

Property (C1). The first closure property is substitutivity of a variable by an access path. Taking the ACC condition as an example, this property is:

$$\mathsf{ACC}_E(E^\Gamma, R) \qquad \text{implies} \qquad \mathsf{ACC}_{E'}(E^\Gamma\delta, R\delta)$$

where $\delta = \{\overline{p} \mapsto \overline{v}\}$ and

$$E = E_1; \overline{v} : S; E_2 \qquad E_1 \vdash \overline{p} : S \qquad E' = E_1; E_2\delta$$

This is not an unexpected property, but requires authors of acceptance criteria to be quite careful.

Let us start by a remark that closure by substitution of terms (and not only access paths) to variables, is verified by typing judgments in just about any reasonable calculus: $E \vdash J$ implies $E' \vdash J\delta$. This property is necessary to prove correctness of type of an application or subject reduction property for β.

In the calculus with modules, substitutivity is used for example to prove correctness of type of a module application. In the presence of module subtyping and satisfaction rules for block types substitutivity is substantially "harder" in some aspects than in a calculus without modules, even though we substitute only a restricted class of expressions, i.e. access paths.

To see the difficulty let us present a simple example. Let E and E', E_1 (names matching (C1) above, E_2 empty) be defined as follows

$$E = E_1; X : \mathsf{Ty}(SIG_X) \qquad E' = E_1 \qquad E_1 = \hat{E}; N : \mathsf{Ty}(SIG_N); \tilde{E}$$

for some $\hat{E}$ and $\tilde{E}$ and the following signatures SIG_X and SIG_N:

```
SIG_X = Sig                          SIG_N = Sig
          Symbol                               Definition f : nat → nat := [x:nat]x
            g : nat → nat                      Symbol
            f : nat → nat                        g : nat → nat
          Rules                                Rules
            g (f 0) ⇒ f 0                        g 0 ⇒ 0
        End                                  End
```

which in the formal syntax is

$$SIG_X = \mathsf{Sig}\ \ \mathsf{g}, \mathtt{f} : \mathsf{Ty}(\mathsf{RewSpec}(E_{gf}, R_{gf}))\ \ \mathsf{End}$$
$$SIG_N = \mathsf{Sig}\ \ \mathtt{f} : \mathsf{Eq}([x{:}nat]x : nat \to nat);\ \ \mathsf{g} : \mathsf{Ty}(\mathsf{RewSpec}(E_g, R_g))\ \ \mathsf{End}$$

where

$$E_{gf} = g : \mathsf{Ty}(nat \to nat);\ f : \mathsf{Ty}(nat \to nat) \qquad R_{gf} = \epsilon\ \mid\ g\ (f\ O) \longrightarrow f\ O\ :\ nat$$
$$E_g = g : \mathsf{Ty}(nat \to nat) \qquad R_g = \epsilon\ \mid\ g\ O \longrightarrow O\ :\ nat$$

Let us show that the substitution $\delta = \{X \mapsto N\}$ is legal or, in other words, that the judgment $E_1 \vdash N : \mathsf{Ty}(SIG_X)$ holds.

Indeed, in the environment E_1 enriched with the contents of the signature SIG_N, we can show that $g\ (f\ O) \approx f\ O$ using the fact that f is specified to be equal $[x{:}nat]x$. Next, using the rule **(RewSpec/Sat)**, we can show

$$E_1;\ f : \mathsf{Eq}([x{:}nat]x : nat \to nat);\ g : \mathsf{Ty}(\mathsf{RewSpec}(E_g, R_g))\ \vdash\ g, f : \mathsf{RewSpec}(E_{gf}, R_{gf})$$

which by the **(Sig/Sub)** gives $E_1 \vdash SIG_N <: SIG_X$. Now it is enough to apply the rules **(Subsumption)** and **(Ty/Sat)** to get $E_1 \vdash N : \mathsf{Ty}(SIG_X)$ as expected.

In this context, by property (C1), $\mathsf{ACC}_E(E^\Gamma, R)$ should imply $\mathsf{ACC}_{E'}(E^\Gamma\delta, R\delta)$ for any E^Γ and R. In particular if R is defined to contain one rule

$$h\ (X.\mathsf{g}\ O) \longrightarrow S\ O$$

which is both confluent and terminating, $R\delta$ becomes non-confluent because of the non-joinable critical pair $h\ O = S\ O$. The counter-example to termination preservation can also be found very simply. It is enough that SIG_X contains the rewrite rule $f\ (g\ x) \longrightarrow g\ (f\ x)$, and SIG_N its opposite, i.e. $g\ (f\ x) \longrightarrow f\ (g\ x)$. Given R equal

$$h\ (X.f\ (X.g\ x)) \longrightarrow h\ (X.g\ (X.f\ x))$$

it is both confluent and terminating, unlike $R\delta$ which is non-terminating.

The simple means to get rid of at least confluence problems provoked by the condition (C1) is to require that the left-hand sides of rules in R contain only symbols defined in E^Γ. The exception could be made for constructors of inductive types of course. This restriction would make R *mutually orthogonal* with all other rewriting systems defined in the environment and its future extensions.

Property (C2). The second closure property requires the acceptance criteria to be closed by environment extension.

$$\mathsf{ACC}_E(E^\Gamma, R) \qquad \text{implies} \qquad \mathsf{ACC}_{E'}(E^\Gamma, R)$$

where

$$E = E_1; E_3 \qquad\qquad E' = E_1; E_2; E_3$$

Closure by environment extension holds for typing judgments in most formal systems and is absolutely required for any system of "loadable environment fragments", ready to use without rechecking: $E \vdash J$ implies $E' \vdash J$. In particular, the system of separately checked theory files, as implemented in Coq until version 7.3 also required this property to be fully flexible.[3]

For definitions by rewriting, this property simply means *modularity*. Even though modularity is far from being trivial for rewriting [77, 39], this property cannot be avoided in any proof assistant permitting separate checking of fragments of developments. Note that at least for confluence, modularity is ensured by orthogonality proposed for the condition (C1).

Properties (C3), (C4), (C5). These properties require that acceptance criteria should be closed by module removal operations: flattening of signatures, functor removal and elimination of manifest modules.

In other words they say that acceptance criteria should not be "aware" of modules. It is actually hard to think of sensible criteria verifying the first two properties but not this one. Conditions breaking them are judgments about modules themselves and not their term or block components, or something like "there is no more than one symbol defined by rewriting in the module X", but it is hard to see why anyone would need this kind of conditions.

Example. We show that there exist acceptance criteria for the calculus of constructions with inductive types and definitions by rewriting which satisfy our closure properties. As stated before it is enough to show that these criteria imply decidability and consistency of the calculus without modules and without manifest blocks.

A positivity condition for inductive types $\mathsf{POS}(E^I := E^C)$ can be found for example in [24]. It limits the syntactic form of kinds for inductive types from E^I and types of constructors from E^C:

- kind of an inductive type must be a product whose co-domain is a sort,
- type of a constructor must be a product whose co-domain is an instance of an inductive type from E^I and whose arguments contain only strictly positive occurrences of inductive types from E^I.

This condition obviously satisfies closure properties (C1)-(C5). It even satisfies general substitutivity (by terms) as inductive definitions are usually studied as anonymous entities [80, 100, 94].

[3] Actually, due to universe constraints this does not hold in Coq, but since verifying universe consistence can be done quickly, it is verified after every module and file operation.

The acceptance condition for rewrite rules, $\mathsf{ACC}(E^\Gamma, R)$, can be based on a modular termination criteria such as Higher Order Recursive Path Ordering [99] or the General Schema [12]. Both these termination criteria, denoted by TC in the sequel, are based on a partial order on the set of function symbols, called precedence, and guarantee the termination of the relation $\longrightarrow_R \cup \longrightarrow_\beta$ for every accepted rewriting system R. Both these criteria are closed by extensions of the set of function symbols, provided that the new precedence is an extension of the old one. The acceptance criterion $\mathsf{ACC}(E^\Gamma, R)$ is defined as follows:
A definition by rewiring (E^Γ, R) is accepted in the environment E iff the following conditions hold:

1. *left-hand sides of rules in R are linear and algebraic, and contain only function symbols from E^Γ, constructors from E and local rule variables,*
2. *R is accepted by TC with the precedence "$E < E^\Gamma$", meaning that*
   - *all function symbols (valid term paths) from E are incomparable with each other,*
   - *symbols in E are smaller than variables in E^Γ,*
   - *symbols from E^Γ are ordered by some ordering $<_{E^\Gamma}$,*
3. *the critical pairs of R are joinable (by R).*

The first condition is very restrictive. In practice it means that all related definitions by rewriting are put in one place. For example if addition on natural numbers is defined in one module, and multiplication in the other, the latter definition cannot contain the distributivity law presented as a rewrite rule.

To explain the correctness of the above criterion, let us start by showing that it satisfies our closure properties.

Condition 1 obviously satisfies properties (C2)-(C5). It does also satisfy (C1), because a path substitution can only influence constructors. Fortunately Lemma 8.1.5 shows that one can derive that a sequence of access paths is an inductive definition only if it has been defined as an inductive definition in the environment and Lemma 8.1.4 shows that this property is closed by substitution. Hence, a constructor remains a constructor after a substitution.

Condition 2 satisfies (C1), because there is no assumption about the relative order of symbols in E, so any substitution has no effect on the comparisons between symbols. The property (C2) follows by the fact that TC is stable by ordering extension. Properties (C3)-(C5) are satisfied trivially.

Condition 3 satisfies all closure properties, because all confluence diagrams for critical pairs remain valid after substitution.

Given a correct environment E, the termination of the relation $\longrightarrow_\beta \cup \longrightarrow_{R_E}$, where R_E is the set of all rewrite rules in E, follows from the properties of TC. Indeed, the way a correct environment is built by adding subsequent rewriting systems corresponds to the notion of modularity from [12] and [99].

Since TC guarantees termination of $\longrightarrow_{R_E} \cup \longrightarrow_\beta$, confluence of $\longrightarrow_{R_E}$ follows from the fact that it is locally confluent. Indeed, all critical pairs issued from a single block are joinable by assumption, and no critical pairs exist between blocks. Confluence of $\longrightarrow_{R_E} \cup \longrightarrow_\beta$ can be proved using the results from [74]. In the end we get a decidability of convertibility which results in a typing algorithm.

The problem of consistency for the calculus of constructions with rewriting has not been very well studied yet. The only condition that we know that guarantees consistency [12] is not modular. However, it is believed that *completeness of definitions* as defined in [58, 27, 15, 18], requiring that all constructor instances of a function symbol are reducible, can be extended to the calculus of constructions and proved to guarantee consistency. Such a condition $\mathsf{COMP}(E^\Gamma, R)$ would certainly satisfy closure properties (C1)-(C5), as constructors do not change under substitutions.

Chapter 12

Conclusions and future work

12.1 Conclusions

We have presented an ML-style module system for a proof assistant based on type theory. In our formalism modules are named entities and therefore well-suited to contain generative definitions such as inductive types and definitions by rewriting. Given modular acceptance criteria for these definitions, we have shown that the typing system with modules is a conservative extension over the typing system without modules and that decidability of the latter implies decidability of the former.

Our theoretical work is parameterized by criteria of acceptance of inductive definitions and definitions by rewriting. We have formulated 5 closure properties which must be satisfied by these criteria in order to make our proofs work. These properties may be shortly described as modularity and closure by substitutions of names to names. They obviously hold for positivity conditions for inductive types (as described in [80, 100, 94, 24]) and are reasonable for termination and confluence criteria for definitions by rewriting. We give a proposition of such criteria in Chapter 11, but finding really flexible criteria satisfying our closure properties is left as a future work.

We have successfully implemented our module system in the released version 7.4 of Coq. As the theory shows, the future planned extension of Coq with rewriting will not conflict with modules.

The modules have already been successfully used in several examples. In combination with the extraction mechanism capable to produce Ocaml modules from Coq modules (implemented by Pierre Letouzey), Coq may become a very convenient tool to develop certified parametric data structures. For example, a modular formalization of certified dictionaries implemented by binary search trees was realized by Pierre Casteran (contribution `Bordeaux/dictionaries`).

Jean-Christophe Filliâtre and Pierre Letouzey have certified [43] a parametric realization of red-black balanced binary trees and the version of AVL trees implemented in the Ocaml Standard Library. A long-standing bug in the balancing of AVL trees has been found and corrected thanks to this formal development.

Another interesting example of using modules is KRAKATOA [68], a Coq frontend to certify JAVA programs. The authors prove quite a few generic properties of Java classes abstractly in a functor, and a user working on a concrete Java program simply applies the functor to a module representing the classes of his program and can conveniently use the properties already instantiated to his particular setting.

12.2 Future work

The immediate continuation of the work presented in this thesis in polishing the implementation of modules in Coq. Other tasks include solving practical problems related to the implementation of rewriting and extending the module system.

Polishing the implementation. Even though the implementation of modules is fully usable, there are still some corrections to be made. The first one concerns error messages which are often quite cryptic. It seems necessary to reimplement error reporting for modules in such a way that the messages are clearer and positions of errors are shown to the user.

Another deficiency of the current implementation is incomplete parsing and pretty-printing of module expressions. Indeed, there is no way to enter a whole structure or signature at once.[1] Also printing of structures and signatures is very superficial: only the names and kinds of module components are printed without constant bodies or type information. The problem however is not completely trivial as parsing and pretty-printing are context dependent and require parallel processing of logical (definitions, lemmas, etc.) and extra-logical features (implicit types, pretty-printing and parsing rules etc.)

The last imperfection of the implementation mentioned in this section is the incomplete subtyping for inductive definitions. In theory

$$\begin{gathered}\mathsf{Sig}\ \texttt{bool}, \texttt{true}, \texttt{false} : \mathsf{Ty}(\mathsf{IndSpec}(E^I_{bool} := E^C_{bool}))\ \mathsf{End} \\ <: \\ \mathsf{Sig}\ \texttt{bool} : \mathsf{Ty}(\star);\ \ \texttt{true} : \mathsf{Ty}(bool);\ \ \texttt{false} : \mathsf{Ty}(bool)\ \mathsf{End}\end{gathered}$$

while in the implemented module system it is not the case. The problem comes from the fact, that the term representing an inductive type `bool` is different from the term representing a constant `bool`. Therefore a substitution resulting from the subtyping judgment presented above must change the structure of terms and not only names appearing in terms. This is very unfortunate as the same substitution is used inside and outside the kernel and the fact that it modifies only names simplifies greatly the implementations.

Two solutions are therefore possible: a dirty hack or a complete change of representation of inductive types and constructors, requiring tedious modification of a few hundred places where the current structure of terms is explicitly used.

[1]Of course any module expression can be entered using interactive modules and interactive module types. It is only a matter of user convenience.

New features. The most important feature of modules which is missing compared to Ocaml is a possibility to have module interfaces (signatures) in separate files. This would help doing large proofs in groups of many people, once they agree on the interface of a proof library, one person could go on proving properties from the library and the other could start using the library, much before all the proofs are done.[2] This would require writing a utility verifying that all bodies of components of a big project do match their interfaces. Such a utility would play a role of a linker which links together all related modules of a big program.

Another interesting feature wish, signaled by users, is the possibility to create multifile functors. It should be possible to write the input signature of a functor in one file, the output signature in another, and work with several files containing proofs until they are completed.

Practical problems with rewriting. The obvious continuation of the work presented in this thesis is solving the practical issues arising when putting rewriting and modules together in Coq.

In particular, our decidability proof for the typing problem for the calculus with modules shows a very inefficient algorithm: translate everything to the calculus without modules and without manifest blocks, and solve the problem there (which is assumed to be feasible). It is therefore necessary to find a direct approach, using oriented delta and rewrite reductions to decide convertibility in the calculus with modules and manifest blocks. This is closely related to the problem of finding flexible acceptance criteria for rewrite rules, assuring termination and (some form of) confluence in the presence of delta rules.

It appears that precedence-based termination criteria such as General Schema [12] or Higher-Order Recursive Path Ordering [99] could easily be used to prove termination of the relation $\longrightarrow_R \cup \longrightarrow_\beta \cup \longrightarrow_\delta$. Indeed, the delta reduction is derived from the judgment $E \vdash_P \overline{p} : \mathsf{Eq}(\overline{\phi} : \Psi)$, and necessarily $\overline{\phi}$ is defined before $\overline{p}$ in the environment so it is smaller than $\overline{p}$ in the precedence. Hence the rewrite rule $p^i \longrightarrow_\delta \phi^i$ should be accepted.

The confluence of $\longrightarrow_R \cup \longrightarrow_\delta$ is more complicated though. In the presence of the restriction on function symbols in the left-hand sides of rewrite rules, presented in our proposition for the ACC condition in Chapter 11, delta reduction on symbols defined by rewriting should not be a problem. Given a manifest definition by rewriting $\overline{v} : \mathsf{Eq}(\overline{p} : \mathsf{RewSpec}(E^\Gamma, R))$ the symbols $\overline{v}$ can in the same time δ-reduce to $\overline{p}$ and reduce using the rules in R, resulting in critical pairs of the form

$$e^L[p^j]_q \quad {}_\delta\xleftarrow{q} \quad e^L \quad \longrightarrow_R \quad e^R$$

Since $\overline{p} : \mathsf{RewSpec}(E^\Gamma, R)$, we have $e^L\{\overline{v} \mapsto \overline{p}\} \approx e^R\{\overline{v} \mapsto \overline{p}\}$ and $e^L[p^j]_q \longrightarrow_\delta^* e^L\{\overline{v} \mapsto \overline{p}\}$ and $e^R \longrightarrow_\delta^* e^R\{\overline{v} \mapsto \overline{p}\}$, so under condition that $e^L\{\overline{v} \mapsto \overline{p}\} \approx e^R\{\overline{v} \mapsto \overline{p}\}$ is joinable, so is our critical pair.

Unfortunately δ-reductions for constructor symbols can create non-confluent critical pairs with rewrite rules, so in order to have efficient algorithm for deciding convertibility,

[2]This is of course all possible now, but at the cost of replacing axioms by hand with the proved lemmas.

one must perform reductions modulo δ-equality for constructors. This does not appear to be difficult, but should be studied in more details.

Another interesting topic is rewriting modulo associativity and commutativity. From the theoretical point of view our proofs already cover this case, since we do not assume that the equations issued from rewrite specifications are oriented. Practically, implementing an efficient reduction engine covering rewriting modulo AC and beta reduction requires quite a lot of work.

This research, combined with experiments with the "Rewriting" branch of Coq should also give much insight on the directions in which the module system should be tuned to make the life of rewriting implementors easier.

Weakening the satisfaction rule. One suggestion that already came from the author of the aforementioned experimental branch of Coq, Frédéric Blanqui, is to weaken the satisfaction rule for $\mathsf{RewSpec}(E^\Gamma, R)$. This would make the substitutivity of names (condition (C1)) much easier to achieve, meaning that more rewrite rules would be accepted. The current satisfaction rule, allowing arbitrary conversion between the left and right-hand side of rules in R, seems to make proving confluence more difficult than if only oriented reductions from the left-hand side to the right-hand side were allowed, i.e. if the condition $e^L \approx e^R$ were replaced by $e^L \longrightarrow^* e^R$.

This modification appears to be feasible, but weakening the $\mathsf{RewSpec}$ satisfaction rule would weaken the module subtyping substantially, so there is a trade-off between allowing more "rewriting" or more "subtyping" and the right compromise can only be found through experiments.

Special treatment of completely defined symbols. We showed in Chapter 11, that while considering a rewriting specification $\mathsf{RewSpec}(E^\Gamma, R)$, allowing symbols from outside of E^Γ in the left hand-side of rules in R is problematic as far as the closure property (C1) is considered.

But it is even worse. In presence of module subtyping such rules are dangerous for consistency. Consider the following piece of Coq code:[3]

```
Module Type SigX.
  Parameter a:nat → nat.
End SigX.

Module F[X:SigX].

  Module Type SigM.
    Parameter f:nat → nat.
    Lemma fs : (x:nat)(f (S x))=(1).
    Lemma fa : (x:nat)(f (X.a x))=(0).
  End SigM.
```

[3]This example, together with the final theorem `(0)=(1)` has been checked in the experimental "Rewriting" version of Coq.

```
    Module M : SigM.
      Symbol f : nat → nat.
      Rules [x:nat] { (f O) => (O); (f (S x)) => (1); (f (X.a x)) => (O) }.
      Lemma fs : (x:nat)(f (S x))=(1).
        Intros; Reflexivity. (* using the second rule *)
      Qed.
      Lemma fa : (x:nat)(f (X.a x))=(O).
        Intros; Reflexivity. (* using the third rule *)
      Qed.
    End M.

  End F.

  Module X <: SigX.
    Definition a:=[x:nat](1).
  End X.

  Module N := F X.
```

Now the type of `N.M.fa` is `(x:nat) (N.M.f (X.a x))=(O)` and using the definition of `X.a` and Lemma `N.M.fs`, we can show that `(x:nat) (N.M.f (X.a x))=(1)`.

Note that there is no critical pairs while checking the rewriting system inside `M` and since this system is hidden by the signature constraint, there is no critical pairs after applying `F` to `X` either.

We therefore found a proof of `(O)=(1)` using the rewrite rule `(f (X.a x)) => (O)`. Indeed, this rule is wrong, because it implicitly assumes that `(X.a x)` is `O` while this is not declared in the functor argument signature.

Having seen the examples in Chapter 11 and the one above, it appears necessary to disallow "foreign" symbols in the left-hand side of rewrite rules. On the other hand it is of course desirable to have the possibility of defining rewrite rules with foreign symbols on the left-hand side, since for example many arithmetic functions f could benefit from rewrite rules of the form $f(a + b) \longrightarrow \ldots$ and it is not very convenient to be forced to declare them all in one definition by rewriting, together with the function $+$.

We suspect that the main difference between the example presented above and the rules $f(a+b) \longrightarrow \ldots$ is the fact that $+$ is completely defined. Going back to our example, if `X.a` were completely defined at the point of checking our questionable rule, we would get a critical pair. If this critical pair were non-confluent, the questionable rule would be immediately rejected. And if the critical pair were confluent, the proof of `(O)=(1)` we did would be impossible, and accepting this rule would probably be harmless.

Clearly, more work is needed on completeness of rewriting and in general on definitions by rewriting in an interactive setting.

Proving the rules correct. One approach to proving consistency of the calculus of constructions with rewriting consists in allowing rewrite rules to be added to the system only after the corresponding equality has been proved. This approach seems very reasonable but has not yet been proved to be correct. This is the subject of the ongoing PhD program of Nicolas Oury.

Since this approach concerns adding more rewrite rules to an existing definition it does not immediately fit into our framework. However, the silent assumption that E^{Γ} is non-empty is not really used in our proofs. So adding rules without defining new symbols is not really outside our formalism. But on the other hand, we like to treat our work as a necessary restriction of a hypothetical anonymous system (described in Section 1.6). In that context it is really hard to think about anonymous definitions by rewriting not defining any symbols.

Putting aside these philosophical questions, it also seems possible to treat the proof of equality as an element of the ACC condition verification and therefore accept it in our system provided that the new rule and the old definition by rewriting are in the same module.

Once the approach presented above is proved to be correct, one can consider an interesting syntactic sugar consisting in using functors. First, we define a functor `F` whose input signature `F_input` contains functions defined by many rewrite rules and whose body uses the rewrite rules to make proofs in a convenient way.

```
Module Type F_input.
  Symbol g : nat → nat → nat,  h:nat → nat
  Rules
    g 0 0 => ...
    g (h x) x => ...
    h (g 0 y) 0 => ...
End F_input.

Module F[X:F_input].
  Lemma lem1 : (x:nat) (g (h x) 0) = (h 0).
  ...
End F.
```

Second we define a structure `M` containing function definitions and prove equalities corresponding to the rules in `F_input`.

```
Module M.
  Definition g := plus.
  Definition h := S.
  Lemma rule1 : (g 0 0) = ...
  Lemma rule2 : (x:nat) (g (h x) x) = ...
  Lemma rule3 : (x:nat) h (g 0 y) 0 = ...
End M.
```

And third, we apply the functor `F` to the structure `M`, obtaining lemmas about functions in `M` proved using the rewrite rules defined in `F_input`.

```
Module M_lemmas := F M.
```

Note however, that if the correctness of the output module type of `F` depends on the rewrite rules in `F_input`, the output module type of `M_lemmas` can be incorrect. It can happen very easily in the presence of dependent types, so it would be necessary to check the correctness of the module type resulting from the module application. This would mean a transformation of the current module application rule into

$$\frac{E \vdash p : \mathsf{Funsig}(v : M_1)\ M_2 \qquad E \vdash p' : M_1 \qquad E \vdash M_2\{v \mapsto p'\} : \mathtt{modtype}}{E \vdash p\ p' : M_2\{v \mapsto p'\}}$$

where the third premise verifies that the application yields a legal module type.

In fact the "Rewriting" branch of Coq already realizes this kind of post-application check, verifying critical pairs of rewrite rules generated by the substitution and rejecting the application if the critical pairs are not joinable. But this resulted most from a coincidence rather than a thoughtful decision.

Quotient types. Even though we have two kinds of blocks, for inductive definitions and definitions by rewriting, nothing prevents us from considering other kinds of blocks, such as definitions of quotient types [6, 35, 34].

Formation rules for these new blocks would have to satisfy the same closure properties in order to match the module system. It can be conjectured that in order to successfully go through with our proofs either there should be no satisfaction rule for such blocks or the satisfaction rule should not require more than what can be derived from its conclusion by elimination and computation rules.

An applicative module system. Our module system puts the same syntactic restriction on access paths to those present in [59]. However, one also could think about relaxing this restriction slightly and put the one described in [60]. This would mean allowing access paths to contain path applications, as in `(M.F M.N).P`. A module system implemented in Ocaml allows this kind of access paths, but they are of course only valid for type paths and not for value paths.

From the theoretical point of view proving conservativity with such restriction should not be too difficult. It would be enough to add phase zero to our two phase proof, extracting all path applications occurring in terms and giving each of them a name. This procedure is shown to be correct in [61] and in our case it would yield a structure ready for our phase one: evaluation.

The relative decidability proof also seems to be feasible. The problem lies in the implementation of this module system in Coq especially when extra-logical features are concerned such as implicit arguments, pretty-printing and parsing rules etc. In the current implementation, all these extra-logical objects are pre-computed for all legal module paths. In presence of applicative modules the set of all legal module paths can easily be infinite, so this approach is not possible.

With the relaxed restriction on paths, proving termination or confluence of rewriting could also be more difficult. In particular the set of all rewrite rules derivable from the environment can easily turn out to be infinite.

EML methodology. An interesting possibility opened by addition of modules is the adaptation to Coq of the EML methodology of gradual construction of functional programs [92], which strongly relies on the use of functors. In a proof assistant framework, correctness of refining steps could be established formally, using interactive proofs.

Bibliography

[1] Davide Ancona and Elena Zucca. An algebraic approach to mixins and modularity. In M. Hanus and M. Rodrìguez Artalejo, editors, *5th Intl. Conf. on Algebraic and Logic Programming*, number 1139 in Lecture Notes in Computer Science, pages 179–193, Berlin, 1996. Springer-Verlag.

[2] Andrew Appel and David B. MacQueen. Standard ML of New Jersey. In J. Maluszynski and M. Wirsing, editors, *Programming Language Implementation and Logic Programming, Proceedings of the 3rd Intn'l Symposium*, volume 528 of *Lecture Notes in Computer Science*, pages 1–13, 1991.

[3] Franco Barbanera, Maribel Fernández, and Herman Geuvers. Modularity of strong normalization in the algebraic-λ-cube. *Journal of Functional Programming*, 7(6):613–660, 1997.

[4] Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[5] Henk Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 2, pages 117–309. Oxford University Press, 1992.

[6] Gilles Barthe and Herman Geuvers. Congruence types. In H. Kleine Buening, editor, *Proceedings of the Conference on Computer Science Logic*, volume 1092 of *Lecture Notes in Computer Science*, pages 36–51. Springer-Verlag, 1995.

[7] Stefano Berardi. *Type dependence and constructive mathematics.* PhD thesis, University of Torino, Italy, 1990.

[8] Gustavo Betarte. *Dependent Record Types and Algebraic Structures in Type Theory.* PhD thesis, PMG, Dept. of Computing Science, University of Göteborg and Chalmers University of Technology, 1998.

[9] Gustavo Betarte. Proof reutilization in Martin-Löf's logical framework extended with record types and subtyping. Technical Report 00-09, InCo-Pedeciba, September 2000.

[10] Gustavo Betarte. Type checking dependent (record) types and subtyping. *Journal of Functional Programming*, 10(2):137–166, March 2000.

[11] Frédéric Blanqui. *Théorie des Types et Récriture*. PhD thesis, Université Paris-Sud, 2001.

[12] Frédéric Blanqui. Definitions by rewriting in the Calculus of Constructions. *Mathematical Structures in Computer Science*, 2003. To appear.

[13] Frédéric Blanqui, Jean-Pierre Jouannaud, and Mitsuhiro Okada. The Calculus of Algebraic Constructions. In P. Narendran and M. Rusinowitch, editors, *10th International Conference on Rewriting Techniques and Applications*, volume 1631 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer-Verlag.

[14] Frédéric Blanqui, Jean-Pierre Jouannaud, and Mitsuhiro Okada. Inductive data type systems. *Theoretical Computer Science*, 272(1–2):41–68, 2002.

[15] Adel Bouhoula and Jean-Pierre Jouannaud. Automata-driven automated induction. In *Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 14–25, Warsaw, Poland, June 1997. IEEE Comp. Soc. Press.

[16] Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. In M. Broy and C. B. Jones, editors, *Proceedings IFIP TC2 Working Conference on Programming Concepts and Methods*, pages 479–504. North-Holland, 1990. Also available as research report 56, DEC Systems Research Center.

[17] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[18] Jacek Chrząszcz. Complétude de systèmes de réécriture d'ordre supérieur. Université de Paris-Sud, July 1997. Mémoire de stage de DEA.

[19] Jacek Chrząszcz. Implementation of modules in the Coq system. In D. Basin and B. Wolff, editors, *Proceedings of the Theorem Proving in Higher Order Logics 16th International Conference*, volume 2758 of *Lecture Notes in Computer Science*, pages 270–286, Rome, Italy, September 2003.

[20] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[21] Horatiu Cirstea and Claude Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, May 2001.

[22] Horatiu Cirstea, Claude Kirchner, and Luigi Liquori. The Rho Cube. In Furio Honsell, editor, *Foundations of Software Science and Computation Structures,* `http://www.disi.unige.it/etaps2001` *ETAPS'2001*, `http://www.springer.de/comp/lncs/index.html`Lecture Notes in Computer Science, pages 166–180, Genova, Italy, April 2001. Springer-Verlag(c).

[23] The Coq proof assistant. `http://coq.inria.fr/`.

[24] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V7.4*, 2003.

[25] Catarina Coquand. Agda, 2000. `http://www.cs.chalmers.se/~catarina/agda/`.

[26] Thierry Coquand. An analysis of Girard's paradox. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 227–236, Cambridge, Massachusetts, USA, June 1986. IEEE Comp. Soc. Press.

[27] Thierry Coquand. Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 71–83, Båstad, Sweden, 1992.

[28] Thierry Coquand and Jean H. Gallier. A proof of strong normalization for the Theory of Constructions using a Kripke-like interpretation. In G. Huet and G. Plotkin, editors, *Preliminary Proceedings of the First International Workshop on Logical Frameworks*, pages 479–497, Antibes, France, 1990.

[29] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, February 1988.

[30] Thierry Coquand, Robert Pollack, and Makoto Takeyama. A logical framework with dependently typed records. In *Typed Lambda Calculus and Applications, TLCA'03*, volume 2701 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[31] Judicaël Courant. A Module Calculus for Pure Type Systems. In *Typed Lambda Calculi and Applications 97*, volume 1210 of *Lecture Notes in Computer Science*, pages 112–128. Springer-Verlag, 1997.

[32] Judicaël Courant. *Un calcul de modules pour les systèmes de types purs*. Thèse de doctorat, Ecole Normale Supérieure de Lyon, 1998. `http://www.lri.fr/~jcourant/publis/liste_publis.en.html`.

[33] Judicaël Courant. $\mathcal{MC}_2$: A Module Calculus for Pure Type Systems. Research Report 1292, LRI, September 2001.

[34] Pierre Courtieu. Normalized Types. In Laurent Fribourg, editor, *Proceedings of the Annual Conference of the European Association for Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 554–569, Paris, France, September 2001. Springer-Verlag.

[35] Pierre Courtieu. *Représentation d'algèbres non libres en théorie des types*. PhD thesis, Université Paris-Sud, 2001.

[36] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 50–63, 1999.

[37] Haskell B. Curry and Robert Feys. *Combinatory Logic I.* North-Holland, Amsterdam, 1958.

[38] N.G. de Bruijn. A survey of the project Automath. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*. Academic Press, 1980.

[39] Nachum Dershowitz. When are two rewrite systems more than none? *Lecture Notes in Computer Science*, 1295:37–??, 1997.

[40] Derek Dreyer, Karl Crary, and Robert Harper. Toward a practical type theory for recursive modules. Technical Report CMU-CS-01-112, Carnegie Mellon University, 2001.

[41] Francisco Durán and José Meseguer. An extensible module algebra for Maude. In *Second International Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[42] Jean-Christophe Filliâtre. Design of a proof assistant: Coq version 7. Research Report 1369, LRI, Université Paris Sud, October 2000.

[43] Jean-Christophe Filliâtre and Pierre Letouzey. Formalization of a finite sets library in Coq, Jul 2003. `http://www.lri.fr/~filliatr/fsets/index.en.html`.

[44] Herman Geuvers and Mark-Jan Nederhof. A simple modular proof of the strong normalization for the calculus of constructions. *Journal of Functional Programming*, 1(2):155–189, 1991.

[45] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris 7, 1972.

[46] Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, January 1992.

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

[48] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In POPL'94 [87], pages 123–137.

[49] Robert Harper and Frank Pfenning. A module system for a programming language based on the LF logical framework. *Journal of Logic and Computation*, 1(8), 1998.

[50] Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. In Daniel Le Métayer, editor, *ESOP*, volume 2305 of *Lecture Notes in Computer Science*, pages 6–20, April 2002.

[51] Tom Hirschowitz, Xavier Leroy, and Joe B. Wells. On the implementation of recursion in call-by-value functional languages. Research report 4728, INRIA, 2003.

[52] The ISABELLE system. `http://isabelle.in.tum.de/`.

[53] Jean-Pierre Jouannaud and Mitsuhiro Okada. Executable higher-order algebraic specification languages. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 350–361, Amsterdam, The Netherlands, July 1991. IEEE Comp. Soc. Press.

[54] Jean-Pierre Jouannaud and Mitsuhiro Okada. Abstract data type systems. *Theoretical Computer Science*, 173(2):349–391, February 1997.

[55] Jean-Pierre Jouannaud and Albert Rubio. The higher-order recursive path ordering. In Giuseppe Longo, editor, *Fourteenth Annual IEEE Symposium on Logic in Computer Science*, Trento, Italy, July 1999. IEEE Comp. Soc. Press.

[56] Florian Kammüller. Modular reasoning in Isabelle. In *17th International Conference on Automated Deduction, CADE-17*, volume 1831 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2000.

[57] Florian Kammüller and Lawrence C. Paulson. A formal proof of Sylow's first theorem – an experiment in abstract algebra with Isabelle HOL. *Journal of Automated Reasoning*, 23(3–4), Nov 1999.

[58] Emmanuel Kounalis. Completeness in data type specifications. In *Proceedings of EUROCAL 85*, volume 204 of *Lecture Notes in Computer Science*, pages 348–362. Springer-Verlag, April 1985.

[59] Xavier Leroy. Manifest types, modules, and separate compilation. In POPL'94 [87], pages 109–122.

[60] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Conference Record of the 22nd Symposium on Principles of Programming Languages*, pages 142–153, San Francisco, California, January 1995. ACM Press.

[61] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996.

[62] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[63] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1997.

[64] Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990.

[65] David B. MacQueen. Modules for Standard ML. In *Proceedings of the 1984 ACM Conference on LISP and Functional Programming*, pages 198–207. ACM Press, 1984.

[66] David B. MacQueen. Using dependent types to express modular structures. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 277–286, St. Petersburg, FL, January 1986.

[67] David B. MacQueen and Mads Tofte. Semantics for higher order functors. In *European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 409–423, April 1994.

[68] Claude Marché, Christine Paulin, and Xavier Urbain. The Krakatoa tool for JML/Java program certification. Available at `http://krakatoa.lri.fr`, 2003.

[69] The Maude system. `http://maude.cs.uiuc.edu/`.

[70] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[71] John Mitchell, Sigurd Meldal, and Neel Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 270–278. ACM SIGACT and SIGPLAN, ACM Press, 1991.

[72] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[73] Moscow ML. `http://www.dina.kvl.dk/~sestoft/mosml.html`.

[74] Fritz Müller. Confluence of the lambda calculus with left-linear algebraic rewriting. *Information Processing Letters*, 41(6):293–299, 1992.

[75] Tobias Nipkow and Christian Prehofer. Type reconstruction for type classes. *Journal of Functional Programming*, 5(2):201–224, 1995.

[76] Objective Caml. `http://caml.inria.fr/`.

[77] Enno Ohlebusch. On the modularity of termination of term rewriting systems. *Theoretical Computer Science*, 136:333–360, 1994.

[78] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[79] Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1997.

[80] Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J. F. Groote, editors, *Proceedings of the 1st International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345, Utrecht, The Netherlands, March 1993. Springer-Verlag.

[81] Christine Paulin-Mohring. *Définitions inductives en théorie des types d'ordre supérieur*. Thèse d'habilitation, Ecole Normale Supérieure de Lyon, 1996.

[82] Lawrence C. Paulson. Isabelle: the next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*. Academic Press, 1990.

[83] Frank Pfenning. Elf: A language for logic definition and verified meta-programming. In *Proc. 4th IEEE Symp. Logic in Computer Science*, pages 313–322, 1989.

[84] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon D. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.

[85] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In *Proceedings of Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[86] Robert Pollack. Dependently typed records in type theory. *Formal Aspects of Computing*, 13:386–402, 2002.

[87] *Conference Record of the 21st Symposium on Principles of Programming Languages*, Portland, Oregon, 1994. ACM Press.

[88] The PVS specification and verification system. `http://pvs.csl.sri.com/`.

[89] Claudio V. Russo. *Types For Modules*. LFCS thesis ECS-LFCS-98-389, University of Edinburgh, 1998.

[90] Claudio V. Russo. First-class structures for Standard ML. *Nordic Journal of Computing*, 7(4):348–374, 2000.

[91] Claudio V. Russo. Recursive structures for Standard ML. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, pages 50–61, Florence, Italy, September 2001.

[92] Donald Sannella and Andrzej Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9:229–269, 1997.

[93] Paula Severi and Erik Poll. Pure type systems with definitions. In *Logical Foundations of Computer Science'94*, volume 813 of *Lecture Notes in Computer Science*, pages 316–328. Springer-Verlag, 1994.

[94] Milena Stefanova. *Properties of Typing Systems*. PhD thesis, University of Nijmegen, 1999.

[95] William W. Tait. Infinitely long terms of transfinite type. In J.N. Dummett M.A.E. Crossley, editor, *Formal Systems and Recursive Functions, Studies in Logic and the Foundations of Mathematics*, pages 176–185. North-Holland, 1965.

[96] Jan Terlouw. Een nadere bewijstheoretische analyse van GSTTs. Manuscript, University of Nijmegen, Netherlands, 1989.

[97] Diedrik T. van Daalen. *The language theory of AUTOMATH*. PhD thesis, Technical University Eindhoven, Eindhoven, The Netherlands, 1980.

[98] Daria Walukiewicz-Chrząszcz. Termination of rewriting in the calculus of constructions. *Journal of Functional Programming*, 13(2):339–414, 2003.

[99] Daria Walukiewicz-Chrząszcz. *Termination of Rewriting in the Calculus of Constructions*. PhD thesis, Warsaw University and University Paris XI, 2003.

[100] Benjamin Werner. *Méta-théorie du Calcul des Constructions Inductives*. PhD thesis, Université Paris 7, 1994.