JFP **17** (3): 287–352, 2007.
doi:10.1017/S0956796806005867 First published online 8 January 2007 Printed in the United Kingdom

$\mathscr{MC}_2$ *A module calculus for Pure Type Systems*

JUDICAËL COURANT

VERIMAG, UMR CNRS 5104, Centre Équation – 2, avenue de Vignate, 38610 Gières, France
(*e-mail:* Judicael.Courant@imag.fr)

Abstract

Several proof-assistants rely on the very formal basis of Pure Type Systems (PTS) as their foundations. We are concerned with the issues involved in the development of large proofs in these provers such as namespace management, development of reusable proof libraries and separate verification. Although implementations offer many features to address them, few theoretical foundations have been laid for them up to now. This is a problem as features dealing with modularity may have harmful, unsuspected effects on the reliability or usability of an implementation.

In this paper, we propose an extension of Pure Type Systems with a module system, $\mathscr{MC}_2$, adapted from SML-like module systems for programming languages. This system gives a theoretical framework addressing the issues mentioned above in a quite uniform way. It is intended to be a theoretical foundation for the module systems of proof-assistants such as Coq or Agda. We show how reliability and usability can be formalized as metatheoretical properties and prove they hold for $\mathscr{MC}_2$.

1 Introduction

Bodies of formal mathematics developed in proof assistants are larger and larger: thus during the last decade, the size of developments in the proof assistant Coq grew by an order of magnitude (from about ten thousands lines long to about a hundred thousands).

As the size of their formal developments grows, users need many features for structuring them:

1. The proof assistant should have a notion of theory, facilitating the development of proof libraries.
2. It should be possible to verify a given development without rechecking the libraries it depends on. Moreover, in order to help team work, it should be possible to check this development even if the libraries it depends on have not been written yet — only their interfaces should be needed.
3. It should support a notion of parameterized theory. For instance, one should be able to develop the theory of (abstract) groups and instantiate it on the set of integers.

4. It should provide some help for finding one's way in existing libraries. For instance, during the course of an interactive proof, the proof assistant should be able to list all lemmas whose conclusions match the current goal. In a tactic-based prover, it should list the applicable tactics.
5. It should offer some theory reasoning support. For instance, when developing the ring theory, one would like to teach the proof assistant how to reduce an expression involving additions and multiplications; when dealing with an order, the proof assistant should be instructed to do transitivity steps by itself, *etc.*

Very roughly, we can classify these features into two categories:

Language support Features needed for giving new definitions or stating new theorems, in other words, features that crucially depend of some language support: proof libraries, parameterized theory, and ability to verify developments separately.

Proof support Features that helps proving theorems but are not needed for stating them and structuring them into theories. Theory reasoning and help fall in this category. Notice that proof support is often build upon some language support.

We are interested in the soundness and conservativity issues these features raise: can we ensure modularity features do not lead to inconsistencies? Do they make more theorem provable or do they just help proving them more easily?

Answering this question is not as easy as one might think for several reasons:

- Proof assistants have developed numerous features for supporting large developments.
- Modularity constructs are intrinsically subtle. For instance, although Moscow ML's modules are based on a sound system (Russo, 1998), the minor extensions present in the implementation render it unsound (Dreyer, 2002; Dreyer *et al.*, 2002).

1.1 Scope and contribution of this paper

The aim of this article is to bring some insight to these modularity issues. As we are rather concerned with the logical issues modularity involves, we restrict ourselves to the language support for the following reasons:

- It is the most fundamental layer as proof support is generally build upon it.
- Few results exists about soundness issues for the language support.
- Although proof-support certainly raises lots of usability questions, it is less problematic with respect to logical soundness: in tactic based proof assistants all tactics are obtained by composing sound elementary tactics, and in other assistants proving the validity of a given decision procedure with respect to a given theory is quite a well-known issue.

The context of our study is type-theory, more precisely Pure Type Systems (Barendregt, 1993), initially introduced by Terlouw and Berardi. These systems are well-suited for expressing specifications and proofs and they are the basis of several proof assistants (Coq, n.d.; Lego, n.d.; Hallgren, n.d.; Twelf, n.d.).

Structuring large developments has been studied in programming languages for a long time. Therefore, the fruitful Curry-Howard isomorphism between functional programming languages and proof languages suggests that we look at the way programming languages structure large developments. The most powerful devices for structuring large programs are module systems — among which SML-like systems are the most powerful — and object-oriented languages. We chose to deal with module systems since they are much better understood theoretically than object-oriented languages. Moreover, object-oriented languages are mostly imperative and it is unclear yet how well objects can fit in the functional programming paradigm.

So, we define an SML-like kernel module system, called $\mathscr{MC}_2$, suitable for proof development. As our emphasis is on modules, and not any specific proof language, we define it over the generic framework of Pure Type Systems.

We believe our work enjoys the following original points:

- We *formalize*, as mathematical properties, issues related to the development of large proofs such as independence with respect to the implementation, horizontal compatibility, upward compatibility, and composability. Other proof systems address some of these, these issues are rarely explicitly stated and even less formalized.
- We present a module calculus, $\mathscr{MC}_2$, for which we proved these properties hold.
- We proved the subject-reduction property, the Church-Rosser property, strong normalization — which implies its logical consistency. As far as we know, most of the other works on modularity here are formally defined, and some theoretical results are claimed and proved for some of them, but none has a proof of consistency (except Pollack's records, see section 2.2.3).
- These properties let us prove that $\mathscr{MC}_2$ is conservative: in other words, it does not let you prove more theorems — but is might let you prove them more easily. As far as we know, only one other work has a similar result (Cardelli, 1997) but in the case of a module system without parameterized modules.

$\mathscr{MC}_2$ is a step towards a theoretical foundation for the module system of Agda (Coquand, 2000) and directly inspired the recent module system of Coq.

In the remainder of this section, we describe three features $\mathscr{MC}_2$ accounts for.

1.1.1 Proof libraries

An highly expectable feature when developing large proofs is a way to have proof libraries. One could expect to make a particular proof mostly by getting the right proof components off-the-shelves.

However, this raises the issue of *compatibility* of proof components one with each other, which is not trivial: given any two (correct) libraries can you guarantee that they can both be loaded in the same proof development? In the current version of Coq, the answer is no[1]. In the following, we call this issue *horizontal compatibility*.[2]

[1] See (Asperti, 1999) for an example of problem.

[2] This issue is in practice well known to LaTeX users: conflicting LaTeX packages are part of their daily nightmares.

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

A particular horizontal compatibility issue is that of namespace management. Indeed, it is often difficult to find a new meaningful name for each theorem. If two proof libraries define theorems with the same name, how does the proof assistant react?

Moreover, one would like to make proofs robust with respect to changes in the proof libraries they use. If `A` is a proof library used by `B`, can we guarantee that once the provider of `A` releases a new version of it (in order to provide more features for instance) the proofs of `B` do not break? We call this issue *upward compatibility*.

More generally, one can wonder whether these issues can be formally stated and whether one can find some system for managing proof libraries that provably address them.

1.1.2 Parameterized Theories

Users would often like to parameterize a whole theory by some abstract mathematical structure and instantiate it on an actual realization of this structure. For instance, when defining and proving sorting algorithms, it is very convenient to have the whole theory parameterized with a set A, a function $ord : A \to A \to bool$, and axioms stating that ord is reflexive, antisymmetric, transitive, total and decidable. Then, one would like to be able to instantiate this theory over the set of integers equipped with the usual ordering and get the usual sorting functions over list of integers and theorems stating that these functions indeed sort their arguments.

1.1.3 Interfaces

We would like to define a notion of *interface* of a development. Such a notion is desirable for three reasons:

Conceptual issue It would provide a convenient way to describe what is proved in a given proof development. If Alice plans to request Bob to write a proof library she needs, an interface is a good unambiguous formal basis for making a deal.

Separate development Moreover, if Alice's proof assistant understand interfaces, she can begin to develop her own proof while Bob is still implementing the missing library.

Independence with respect to the implementation Finally, interfaces make proofs more robust with respect to changes in the implementations of the libraries they use. For instance, if Alice and Bob agreed on some interface, Bob is free to change anything he wants in his development as long as it still implements the interface. The compliance of Bob's library to the interface should be checked by the proof assistant at the time Bob checks his library, not at the time it is used by Alice. Moreover, this compliance should ensure that the whole development is still correct.

1.2 Plan

Our plan is as follows. In section 2, we present related works and say how they compare to $\mathscr{MC}_2$. In section 3, we present $\mathscr{MC}_2$ through examples developed in

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

our prototype implementation, **oeuf**. In section 4, we give the grammar of $\mathscr{MC}_2$ and its typing rules. Then in section 5, we explain how the expected properties of $\mathscr{MC}_2$ can be formalized as metatheoretical statements about $\mathscr{MC}_2$. Moreover, we sketch the proof method we used to prove them in (Courant, 1999) and give a sound and complete type inference algorithm. In section 6 we describe how the implementation was done. We draw perspectives for future work in section 7 and we make concluding remarks in section 8.

2 Related works

We now present existing approaches to modular proof development.

2.1 User-interaction oriented

2.1.1 IMPS

The Interactive Mathematical Proof System (Farmer *et al*., 1995) is based on salient characteristics of the actual mathematician practice (Farmer *et al*., 1996). This system uses a logic based on higher-logic with partial objects. It is by essence interactive: the focus is not on a *language* for describing theories, but on the *interactive commands* helping to build them and instanciating them.

IMPS promotes the use of little theories (Farmer *et al*., 1992) in a sense very closed to the one used by mathematicians when speaking about set theory, group theory, *etc*. and reminiscent of Bourbaki's work (Bourbaki, 1970). In IMPS, a structure is a tuple of terms enjoying some properties. Application of a theory to a structure is done by giving it the tuple of values constituting the structure. This leads to proof obligations (the properties the tuple must enjoy) that IMPS solves automatically or requests the user to prove interactively.

IMPS helps managing large theories by guiding the user in the course of a proof: when she wants to apply a lemma, IMPS presents her the list of applicable lemmas rather than require her to know its exact name.

A very original characteristic of IMPS is its support for theory reasoning: when developing a theory the user can define tactics, called *macetes*, which can be local to the theory being defined or which can be transported to structures the theory is instanciated on. Macetes are defined in a quite limited language. They are therefore mostly used for simplifying expressions.

It would be interesting to see how the notion of macetes can be added to $\mathscr{MC}_2$. However, since $\mathscr{MC}_2$ and IMPS use quite different framework (for instance, the notion of proof obligations generated by theory applications in IMPS does not fit well in PTS-based proof-assistants), further work is needed. For instance, we would have to add them from the user-interaction or language point of view: do we want macetes to be managed by an interface above $\mathscr{MC}_2$ which would use the available macetes to generate the proof of some subgoals or to be part of $\mathscr{MC}_2$ theoretical framework and metatheory:

- Even in the first case, macetes have to be part of theories. Could they be represented as PTS terms and what would be a suitable representation?

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

Otherwise, could they be added to $\mathscr{MC}_2$ such that the conservativity result we prove in Section 5.2 trivially still holds? (What we are thinking of here would be a kind of structured comments that could be used by an external interface but would have no effect on other constructs of $\mathscr{MC}_2$.) In both cases, how would the interface access representation of macetes in $\mathscr{MC}_2$ for applying them?

- Having general macetes in $\mathscr{MC}_2$ would be more satisfying from a theoretical point of view. However, it would be more difficult since we would have to internalize in $\mathscr{MC}_2$ the notion of tactics. Unfortunately, the metatheoretical status of tactics in type theory is not clear yet, even without modules. Some work by Jacek Chrząszcz is in progress to extend $\mathscr{MC}_2$ with rewriting though.

The large body of mathematics formalized in IMPS raised interesting issues about intertheory reasoning (Farmer, 2000). Some of these issues are quite easily dealt with in $\mathscr{MC}_2$: for instance, extending in place a theory with definitions (a parameterized module in $\mathscr{MC}_2$) is just a matter of adding a new definition in the body of a module and since $\mathscr{MC}_2$ admits subtyping, this addition is correct. Conversely, some issues that are easily dealt with in IMPS might be difficult to handle in $\mathscr{MC}_2$. Consider the case where one user proves some theorem A in a theory T as follows: first she proves a lemma L in T, then she proves a theorem B in a theory U using L *via* an interpretation from T to U and finally, she proves A using B *via* an interpretation from U to T. IMPS allows her to store the theorem A with the theory T whereas in $\mathscr{MC}_2$ one would have first to define a parameterized module T_1 that would provide L, then a parameterized module U that would provide B then another parameterized module T_2 that would provide A: one would need two different modules T_1 and T_2.

The point here is that Farmer's intertheory infrastructure and $\mathscr{MC}_2$ use very different ways to guarantee the consistency of theories:

- Farmer's infrastructure, in the interactive spirit of IMPS, let the user build a valid network of theories from another one *via* some operations that guarantee the preservation of consistency, that is, it relies on the history of the development of the network.
- In $\mathscr{MC}_2$, the proof-checker is given a network of theories that has to be checked for consistency — without any information about the history of its construction. In order to prevent any circularity, mutually dependent modules are forbidden: the dependency graph of modules must be acyclic. Acyclicity is ensured via lexical scoping: a given module can only use previously defined modules.

We can note however that the lack of historical information does not *imply* that the validity of a network with mutually dependent theories cannot be checked, but it seems to make it more difficult. Whether a module language such as $\mathscr{MC}_2$ could admit such a form of mutual dependency between modules while still enjoying separate checking is an open question. Although there is an interest in recursive and mutually defined modules in programming languages (Flatt & Felleisen, 1998;

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

Crary *et al.*, 1999; Hirschowitz & Leroy, 2002), we are no aware of any work on such kind of mutually dependent modules for proofs.

2.1.2 PVS

PVS provides theories similar to Modula-2 style modules (Shankar *et al.*, 1993; Owre *et al.*, 1999; Kammüller, 1996). These theories have been formally defined (Owre & Shankar, 1997), but not formally justified.

PVS allow the definition of parameterized theories. Like for IMPS, the application of a theory to a structure leads to proof obligations that PVS solves or requests the user to prove interactively.

2.2 Language oriented

2.2.1 Isabelle

Kammüller, Wenzel and Paulson have introduced in Isabelle an interesting feature called *locales* (Kammüller *et al.*, 1999).

This feature lets the user define some kind of sets of notations, called proof contexts, containing some constants, axioms and definitions. With these contexts, the user can:

- Open a context. This operation introduces the constants, axioms, definitions and already proved theorem the given locale contains. Isabelle then add the subsequently proved theorems to this locale.
- Export a theorem. This quantifies the proved theorem over the constants of the locale so that it can be instantiated individually at a later time.
- Close a context in order to stop adding results to some context.

Moreover, a locale can be built by augmentation of another locale and several locales can be opened at once (actually there is a stack of currently opened locales).

These locales seem to be a very convenient feature for the user, as they allow her to tell only once the hypothesis she needs in some development. Moreover, a locale that has been closed at some point of a development can later be opened again.

Locales and $\mathscr{MC}_2$ both address the issue of providing a good input language, but very differently:

- Locales in Isabelle are rather concerned with the user (input and output) syntax, that is with providing the user efficient ways to instantiate previously defined theorems easily and providing good syntactic notations. They succeed in this respect. In $\mathscr{MC}_2$, this problem is not addressed for the moment.
- $\mathscr{MC}_2$ addresses the issue of theory parameterization and instantiation whereas locales do not. Instead, Kammüller uses dependent types in addition to locales (Kammüller, 2000).
- From the metatheoretical point of view, $\mathscr{MC}_2$ is a full language needing a metatheoretical justification. On the contrary, locales can be seen just as a kind of evolved macro system, whose semantics is given by translation to a well-known sound language.

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

In fact $\mathscr{MC}_2$ and locales are quite orthogonal concepts, and locales could probably added on top of $\mathscr{MC}_2$. It remains to see however how module instantiation and locales interact. Especially, one may expect problems similar to those introduced by the `include` construct (see Section 3.5.2).

Isabelle has another device for modular reasoning: axiomatic type classes (Wenzel, 2001), similar to Haskell type classes. These type classes exhibit roughly the same advantages and drawbacks as their Haskell counterpart:

- Using them, the proof developer can define very lightweight notations. She can use the same notation over different structures and even tell Isabelle that some definitions can be lifted from some structure to another. For instance, one can have "+" denote addition over integers and have this definition lifted to, say, cartesian products and functions; thus if we have $f : x \mapsto (1, 2 \times x)$ and $g : x \mapsto (5 \times x, 2)$ then $f + g$ denotes the function $x \mapsto (1 + 5 \times x, 2 \times x + 2)$.
- In a given development it is difficult to consider a given type has two different structures. Indeed, given a type and a type class, there is at most one way the type can belong to the class. This means for instance that a given type can be considered a monoid only for one particular monoid structure over this type. In other words, if you want to consider the additive and multiplicative monoid of a given ring, you have to define two distinct type classes. Similarly, you can define the product of two orderings to be the lexicographic ordering or the conjunction of the orderings, but you can not give both definitions and then use one in some part of your development and the other in another part.

These drawbacks mean type classes are incompatible with the horizontal compatibility we mentioned Section 1.1.1. The problem is type classes instance declarations have a non-local effect as they add a property (having some structure) to a previously introduced type. In other words, the binding from a given type to the class it belongs to is not lexically scoped. Designing type classes that would have better properties with respect to modularity seems quite challenging.

2.2.2 Betarte's dependently typed records

Gustavo Betarte and Alvaro Tasistro have proposed to add dependently typed records to type theory in order to formalize some body of abstract algebra (Betarte, 2000b; Betarte, 2000a; Betarte, 1998). One advantage of their system over ours is that records are first-class expressions and functors are ordinary functions working over record types.

However, we believe the lack of manifest fields for their record types has serious drawbacks: the lack of sharing in the language forces the user to develop theories using sharing by parameterization, also known as Pebble-style sharing. Although in the case of *proof* languages the issue Pebble-style sharing *versus* ML-style sharing is too young to be clear-cut, in the case of *programming* languages the debate is over: it is folklore that SML-style sharing is essential for modularity. The arguments given for programming languages seem to apply as well for proof languages: the problem with Pebble-style sharing is you have to know in advance what you will share, which is non-modular. For instance with Pebble-style sharing, when you define partial

orders you have to parameterize them by their carrier set if you want to be able to define the operation "intersection of partial orders"; when you define vector spaces, you have to parameterize them by their field, if you want to define the product of vector spaces, where their field might have to be parameterized by its base set depending on the operation you want to define on it. Pollack presents convincing evidences that ML-style sharing is more adapted for proof languages (Pollack, 2000).

An advantage of these records over $\mathcal{MC}_2$ is their first-class status. On the other hand, this status renders their metatheoretical study more difficult. Although Betarte gives some informal arguments for justifying the rules he proposes (Betarte, 1998), we are not aware of any proof of the logical consistency of his system.

Unfortunately, Betarte's dependently typed records are not expressive enough for being used as module interfaces, as we show Section 3.2.1. Thus they do not address issues such as separate compilation, horizontal compatibility or upward compatibility.

2.2.3 Pollack records with manifest fields

To our knowledge, Randy Pollack's dependently typed records (Pollack, 2000) are the closest work to ours: as we do, he promotes manifest fields as a very useful feature for structuring mathematical developments. However, his aim is not to define a module system for proof assistants. Rather he shows that constructs similar to ours can be coded inside the current implementation of Coq (Barras *et al.*, 1999) thanks to sigma types and coercive subtyping. One advantage of this approach is that the the underlying type system of Coq is already known to be consistent and implementing features mentioned in Section 3.5.2 such as signature abbreviations or the `where type` construct is quite straightforward.

However, this approach has several drawbacks:

- The coding relies on coercive subtyping whose theoretical foundations are not well understood yet, despite some recent progress in this area (Jones *et al.*, 1998). Although the semantics of target language of the coding (Coq's Calculus of Inductive Constructions with universes) is well-understood from a metatheoretical point of view, the reduction semantics of the source language is less clear — the Church-Rosser property being especially problematic.
- As Pollack's structures are built over a quite large stack of layers (the Calculus of Constructions, Coq records, coercive subtyping), their type-checking is quite costly.
- It might be difficult to solve user interface issues in a satisfactory way: it would be better if the internal layers were hidden to users, but this might be quite difficult to achieve. By now, the implemented system can only be used by type theory experts.

2.3 Building on the Curry-Howard Isomorphism

2.3.1 Elf

Robert Harper and Franck Pfenning have proposed an SML-like module system for Elf, a logic programming language based on the LF logical framework in (Harper

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

& Pfenning, 1992; Harper & Pfenning, 1998). However, as they wanted to ensure this adaptation was safe, Harper and Pfenning chose to be quite conservative: they decided to keep only the part of the SML module system whose semantics was well-understood at this time. They hence left out a significant fraction of the power of the SML module system. For instance, the sharing equations of SML, similar to the manifest fields feature, had to be ruled out.

This is quite unfortunate from our point of view as manifest fields are a key feature for building parameterized proofs (Pollack, 2000). Actually, adding sharing equations is the first future work mentioned by Harper and Pfenning. We think $\mathscr{MC}_2$ is an interesting foundation for adding such a feature.

2.3.2 Related issues in programming languages

Cayenne and Agda Lennart Augustsson's Cayenne language (Augustsson, 1999; Augustsson, 1998) is an Haskell-like functional programming language with a powerful type system based on dependent types. Cayenne introduces records and record types similar to translucent sums. Moreover, Cayenne provides a hierarchical namespace, similar to the one of Java packages. As Cayenne has dependent types and unrestricted recursion, type-checking of Cayenne is undecidable but Augustsson claims this is not a problem in practice. Although presented as a functional language, Cayenne is at the border between functional programming and proof systems: actually, the proof system Agda (Coquand, 2000; Coquand & Coquand, 1999) is an adaptation of Cayenne for type theory.

A significant difference of Cayenne and $\mathscr{MC}_2$ is Cayenne has first-class modules whereas $\mathscr{MC}_2$ separates the module level with the base level syntactically. This sacrifice in terms of expressiveness let us manage the metatheory of modules in $\mathscr{MC}_2$ quite independently of PTS terms.

On the contrary, proving metatheoretical results about Agda's or Cayenne first-class modules is more intricate.

Subject-reduction and Church-Rosser property have been conjectured for Agda and Cayenne but no proof has been given yet.

$\mathscr{MC}_2$ and our metatheoretical work about it (Courant, 1999; Courant, 2002b) are a first step towards a formal justification of these systems, and give hints about the potential problems for proving these conjectures:

- The Church-Rosser property does not hold on untyped term in $\mathscr{MC}_2$, in a more critical way than in Church-style lambda-calculi with η-reduction (see Appendix D). If there is any reduction at the type level (and there are in Cayenne and Agda), this means subject-reduction and the Church-Rosser property have to be proved simultaneously.
- The easiest way such a proof can be done seems to be together with strong normalization, as Goguen did in his thesis for UTT (Goguen, 1994). We showed recently that this approach can be applied to singleton types (Courant, 2002b), which are a simpler formalism exhibiting the specificities of reductions within module systems (Stone & Harper, 2000). This is the most promising way for attacking the metatheory of Agda.

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

- The case of Cayenne is more problematic since it is known not to normalize. Goguen's proof technique relies on the notion of semantic objects, defined as strongly normalizing terms having a unique normal form. In order to adapt it to Cayenne, one would have to change the definition of semantic objects to some (possibly infinite) execution trace for terms. Whether the Church-Rosser and subject-reduction properties can be proved in this framework is an open question.

Syntactic approaches Our work has been much inspired by SML-like module systems (MacQueen, 1984; Tofte, 1996). Especially important to us were Harper and Lillibridge's theoretical work on translucent sums (Harper & Lillibridge, 1994; Lillibridge, 1997) and Leroy's on manifest types (Leroy, 1994; Leroy, 1995). Their works both present variants of the SML module system which are more elegant and have better metatheoretical properties than the initial SML module system. For instance they allow true separate compilation since only the knowledge of the type of a module is needed in order to typecheck modules using it.

We first tried to adapt these systems to proof systems. When we tried to prove they were conservative over Pure Type Systems, we wanted to use the subject-reduction property. This property is critical for reasoning about a module system. Without it, there is little hope one can give a little-step reduction semantics for modules.

Unfortunately, existing systems did not enjoy the subject-reduction property, nor even the substitution property:

- Leroy's systems, accesses to modules fields are restricted to a syntactic category called *module paths* which is not stable through substitution.
- The syntax of Harper and Lillibridge's terms is stable through substitution but modules appearing in types and type operators (called constructors in (Harper & Lillibridge, 1994)) are also restricted, to a syntactic category called *values*.

To see this, consider an environment Γ of the form $\Delta, x : M$, where M denotes the signature $\texttt{sig}\ b{:}\mathsf{Set}; y{:}b;\ \texttt{end}$. In $\mathscr{MC}_2$, the judgment $\Gamma \vdash x.y : x.b$ holds by the rules SPEC/VAR and SPEC/SELECT. The corresponding judgments in Leroy's as well as in Harper and Lillibridge's systems also hold. Now consider a module m of type M in Δ. The judgment $\Delta \vdash m.y : m.b$ still holds in $\mathscr{MC}_2$, whereas it holds in Leroy's and Harper and Lillibridge's systems only under some conditions:

- In Leroy's system, this judgment can be *stated* only if m is a *module path* (Leroy's module paths are given by the grammar $p ::= v|p(p)|p.w$ where w denotes a field name and v denotes a variable). Otherwise, $m.y$ and $m.b$ are syntactically incorrect.
- Similarly, in Harper and Lillibridge's system, this judgment can be stated only if m is a *value*. Otherwise $m.b$ is syntactically incorrect, although $m.y$ is syntactically correct. One may wonder whether $m.y$ can be typed, but in some cases the answer is negative, as shown in (Harper & Lillibridge, 1994), Section 3.1. Indeed, in Harper and Lillibridge's system, the point is one has to remove the dependency from the declaration of y in M to b before one can type $m.y$. The idea is to remark first that m has type $\texttt{sig}\ b{:}\mathsf{Set} := m.b; y{:}b;\ \texttt{end}$,

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

thanks to the VALUE-O rule, similar to our self rule, and then to use subtyping to show that this latter signature is a subtype of $\texttt{sig}\ b\texttt{:Set} := m.b\texttt{;}\ y\texttt{:}m.b\texttt{;}\ \texttt{end}$ in which no dependency from y to b appears. Unfortunately, the VALUE-O rule is limited to the case m is a value, hence it does not always apply.

- By contrast in $\mathscr{MC}_2$, there is no syntactic restriction on field selection: $m.b$ and $m.y$ are syntactically correct for any syntactically correct module expression. The typing rule SPEC/SELECT applies to all module expressions, without any restriction.

This lack of substitution property can immediately be translated to a counter-example for subject-reduction: one just has to consider the redex

$$((\texttt{functor}\ x\texttt{:}M \to \texttt{struct}\ z\texttt{:=}x.y\texttt{;}\ \texttt{end})\ m)$$

with m of type M not fulfilling Leroy's nor Harper and Lillibridge's conditions. This redex is well-typed but its reduced form $\texttt{struct}\ z\texttt{:=}m.y\texttt{;}\ \texttt{end}$ is incorrect since $m.y$ is not typable.

We analyze this lack of substitution property as follows: Harper and Lillibridge as well as Leroy are interested in giving a type system for a programming language. Since type-checking needs to compare types, if expressions of the programming language appear in types, one has to test equality of code, which blurs the phase distinction (Harper *et al.*, 1990) and is undecidable in usual programming languages. Therefore they put some syntactic restrictions on accesses to module fields, thus losing subject-reduction. As we have shown (Courant, 1997), the alternative choice could be made.

In proof systems based on type theory, testing equality of expressions is decidable. Therefore, removing Leroy's and Harper and Lillibrige's restrictions causes no harm. Moreover, although $\mathscr{MC}_2$ has much more satisfying metatheoretical properties with respect to reduction, the complexity and the number of its rules is quite similar to the Leroy's and Harper and Lillibridge's systems.

One interesting question is whether one can have the advantages of both approaches in the case of programming languages: it would be interesting to try to give a system that would have no restriction on accesses to module fields and would use an approximation of the equality test for code. Since β-equivalent expressions must be considered equal for having subject-reduction, this test would be an optimistic approximation of the equality: for instance, when given two expressions of the programming language, it could just consider they are equal.

Singleton types and kinds Harper and Stone recently studied languages with singleton kinds (Stone & Harper, 2000). Singleton kinds are intended to model manifest types. Harper and Stone have a decidability result for equality in their language, but have not given a reduction semantics so far. Using our work on $\mathscr{MC}_2$, we could define reduction notion for a lambda-calculus with singletons types and could adapt our proof of subject-reduction, Church-Rosser property and strong normalization for $\mathscr{MC}_2$ (Courant, 1998; Courant, 1999) to it (Courant, 2002b).

Russo's static semantics Russo recently proposed another approach for modularity (Russo, 1998). Russo's approach relies on a static semantics for SML modules,

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

replacing the need for (a kind of) first-order dependent types in the previously cited works by a simpler second-order dependency.

This simplification allows him to define several extensions such as higher-order modules and first-class modules. The only price to pay seems that in his module system, semantic module types (module types as inferred by the type-checker) are quite different from the syntactic module types (the type entered by the programmer).

However, although Russo's system is well-suited to programming language, we do not see how to adapt it to languages with dependent types such as proof languages without losing the benefits of his approach. Indeed, in Russo's framework, the static semantics of a signature containing some abstract types is a record existentially quantified over these types. Extending Russo's framework to dependently typed languages would require to existentially quantify this record over all abstract components of the signature. In other words, the second-order quantification involved in Russo's static semantics would turn to first and second order quantification as quantifying over terms (and not only types) would become necessary (see section 3.3.5 of (Russo, 1998)).

3 Informal presentation

In this section, we show informally how our module calculus $\mathscr{MC}_2$ addresses the issues mentionned in the introduction. We assume the reader has a working knowledge of the Calculus of Constructions. No previous knowledge of SML-style module systems is required as $\mathscr{MC}_2$ is introduced step by step. The reader familiar with SML-like module systems will notice however that $\mathscr{MC}_2$ concepts are mostly a transposition of SML modules concepts (Harper, 2002).

The examples given in this paper below have been chosen for the sake of simplicity. They show how to deal with mathematical structures in $\mathscr{MC}_2$, but also the limitations of the second-class modules approach of $\mathscr{MC}_2$: $\mathscr{MC}_2$ lets the user state some generic proposition over, say, preorders, but would not let him quantify over preorders like he could with a type of the base calculus (the Calculus of Constructions). Time will tell whether this is a hard limitation for mathematicians (dealing with sets of preorders or sets of groups is not uncommon).

However, it is much less a limitation when dealing with objects of computer science. As an evidence of this, a substantial case study on AVL trees has been conducted in the subset of $\mathscr{MC}_2$ implemented in Coq (Filliâtre & Letouzey, 2004) and David Pichardie developped a modular theory of lattices which has been used to develop certified static analysis of programs (Cachera *et al.*, 2005). Also Coq's module system has been used for certifying compiler optimization (Bertot *et al.*, 2005); according to the authors, it helped them "factor out a significant part of specifications and correcteness proofs".

All examples given in this article have been checked in our prototype implementation of $\mathscr{MC}_2(CC)$ — $\mathscr{MC}_2$ over the Calculus of Constructions — **oeuf**. In fact, we wrote them and this article at once, using noweb (Ramsey, 2001; Ramsey, 1994; Johnson & Johnson, 1997): the code chunks below can be extracted from the source of this article (available from **oeuf**'s web page at `http://www-verimag.imag.fr/~courant/soft/oeuf/`) using noweb; they can subsequently be checked by **oeuf**.

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

3.1 Structures

In order to solve the problem of namespace management, we add to PTS the notion of *structures*, that is, packages of definitions. A structure is a list of named definitions; it is introduced with the keyword `struct` and is terminated with the keyword `end`. Each definition is composed of an identifier representing the name of the field to be defined, the symbol `:=`, the actual definition and a semicolon.

To be more concrete, let us consider an example: the formalization of preorders. A preorder could be represented by a structure with a set `a` representing its carrier, a relation `<=`, the proof `refl` that `<=` is reflexive, and the proof `trans` that `<=` is transitive. The structure representing the Preorder over natural numbers equipped with the usual order over natural numbers can be written as follows:

⟨*structure of preorder over natural numbers*⟩≡

```
struct
 a := Std.nat;
 ⩽ := Std.le_nat;
 refl := ⟨proof of reflexivity of ⩽⟩;
 trans := ⟨proof of transitivity of ⩽⟩;
end
```

where `nat` is the set of natural numbers and `le_nat` is the usual order over natural numbers.

In our prototype, we can bind this structure to, say, the name ”NatPreorder” by typing:

⟨`NatPreorder` *definition*⟩≡

```
NatPreorder := ⟨structure of preorder over natural numbers⟩
```

From inside the above structure, components are referred to as `a`, `<=`, `refl`, and `trans`. Once this structure is bound to the name `NatPreorder`, these components must be referred to as `NatPreorder.a`, `NatPreorder.<=`, `NatPreorder.refl`, and `NatPreorder.trans` from the outside.

3.2 Signatures

Binding a structure to a name in $\mathscr{MC}_2$ is done through a module definition. In order to address the issue of robustness of proofs with respect to changes, we introduce the notion of structures' signatures. When we want to bind a structure m to a name x, we can declare a signature M as the intended interface for x. Through this interface, the proof developper can express which informations should be exported by the module x. She can for instance choose to hide or to reveal the existence of some fields of m, or to hide or to reveal their contents. We use the syntax $x{:=}m{:}M$ to denote such a declaration. In order to process it, we first check that the structure m implements the interface M, and add to the environment the association $x : M$.

We can also bind a module to a name without giving any interface, like we did for `NatPreorder` in the previous section. In this case, we consider all the informations contained in the structure we bind should be exported. That is, we compute the interface revealing as much information as possible about this structure and take it as the interface of the defined identifier.

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

In both cases, notice we can forget m afterwards as it has no role to play in the processing of the following declarations.

The interfaces we propose have the following interesting features:

- They help structuring development, making clear what the different parts of a given development prove.
- They allow separate checking of theories. In our prototype **oeuf**, the user can check the interface of a theory x and develops a module y depending on x *before* she actually implements x since **oeuf** only needs to know the interface of x in order to check the implementation of y.
- They make proofs more robust with respect to changes. One may decide to change a proof done in some module x. In Coq or LEGO, this may break developments built over x. In **oeuf**, this can only happen if one changes the *interface* of x. In other words, changing the *implementation* of x without changing its *interface* never breaks any development built over it.
- They reduce the amount of memory needed by an actual proof-checker to check a development. In Coq, when one wants to develop a proof depending on some module x, one has to load the whole development of x. That is, one not only loads the statements of theorems proven in x but also the actual proofs of these statements.

3.2.1 *Abstract and manifest fields*

The signature of a module is a list of field declarations, enclosed between the keywords `struct` and `end`. A declaration is a field name, followed by a colon, a specification giving the type of this field, and a semi-colon.

For instance, a possible signature for *NatPreorder* defined above could be:

⟨*generic preorder signature*⟩≡

```
sig
 a : Set;
 ⩽ : a → a → Prop;
 refl : ∀x:a . x ⩽ x;
 trans : ∀x y z :a . x ⩽ y → y ⩽ z → x ⩽ z;
end
```

If we define `NatPreorder` with this interface, we have the problem that from the outside, its field `a` is completely abstract. Actually, that is precisely what an "abstract type" is in programming languages: although `NatPreorder.a` was implemented as the set of natural numbers, our type system refuses to consider `0` as being of type `NatPreorder.a` since nothing tells it that `NatPreorder.a` and `nat` actually denote the same set. Thus the type-checker rejects the expression `0 ‘NatPreorder.⩽‘ 0`.

In order to fix this problem, we allow the specifications of the fields declared in an interface to be *manifest*, telling the type-checker what the value of the field is. Instead of being just a type t, a specification is allowed to be either a type t or the datum of a type t and a term t' being its manifest value. The syntax for a manifest specification is $t := t'$ where t is its type and t' is its value.

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

Thus, a reasonnable interface for the preorder of natural number would declare `a` and $\leqslant$ as manifest in order to be able to introduce elements of type `a` and to prove that some given elements are in relation by $\leqslant$:

⟨*NatPreorder interface*⟩≡

```
sig
 a : Set := Std.nat;
 ⩽ : a → a → Prop := Std.le_nat;
 refl : ∀x:a . x ⩽ x;
 trans : ∀x:a . ∀y:a . ∀z:a . x ⩽ y → y ⩽ z → x ⩽ z;
end
```

Then, we can define `NatPreorder` with this interface as follows:

⟨`NatPreorder` *done right*⟩≡

```
NatPreorder := ⟨structure of preorder over natural numbers⟩
            : ⟨NatPreorder interface⟩
```

The type-checker still considers `NatPreorder.trans` is a normal form but reduces the expression `NatPreorder.a` to `nat` and `NatPreorder.⩽` to `Std.le_nat`. Therefore it accepts `0 ‘NatPreorder.⩽‘ 0`.

This "manifest field" feature makes $\mathscr{MC}_2$ depart fundamentally from formalisms implementing packages as record types (Betarte, 2000b; Tasistro, 1997; Betarte, 1998; Pollack, 1997). Indeed, in these formalisms, there is no middle way: given a record expression e, either it reduces to an explicit record whose value of all fields are known or it reduces to a variable (possibly applied to some arguments) and none of its fields can be known. On the contrary in $\mathscr{MC}_2$ a record expression e can be such that some of its fields reduce to known concrete values and some do not.

3.2.2 *Private fields*

Thanks to interfaces, a developer can also hide definitions and lemmas she considers uninteresting or too strongly depending of the proof method he chooses: in order to do that, she just has to omit them from the signature she declares.

For instance, assume she develops a module about Fermat's lesser theorem.[3] If she chooses to use the well-known proof by induction on a, using the binomial theorem and a technical lemma stating that for all $k < p$, p divides $\binom{p}{k}$, she will probably not export the technical lemma as it is not of general use. Exporting it would even be a bad idea. After the initial release of her proof library, the developer may indeed notice that Fermat's lesser theorem is just a particular instance of Euler's theorem[4] and might change her development to reflect this. If she exported the previous technical lemma, she would have to choose either to remove it and break upward compatibility, or to keep in her development a proof completely unrelated to the rest of the proof method.

[3] Fermat's lesser known theorem states that for all a and p if p is prime and does not divide a then $a^{p-1} \equiv a\ [p]$.

[4] Euler's theorem states $\forall a, n\ a^{\phi(n)} \equiv 1\ [n]$, where ϕ is Euler's totient function; $\phi(p) = p - 1$ for any prime number p

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

3.3 Sub-structures

Structures can even contain sub-structures, which may help in structuring the environment. In fact, many mathematical structures own sub-structures. Thus, the polynomial ring $A[X]$ over a ring A may be defined as a structure having A as a component; a monoid homomorphism may be defined as a structure having the domain and the range monoids as components. A preorder homomorphism is described by its *Domain* and *Range* preorders, together with a function from the carrier of the former to the carrier of the latter, and a proof that this function preserves the ordering of preorders:

⟨*generic preorder morphism signature*⟩≡
```
sig
  (* domain *)
  D : ⟨generic preorder signature⟩;
  (* range *)
  R : ⟨generic preorder signature⟩;
  f: D.a → R.a;
  compat: ∀x y : D.a . x ‘D.≤‘ y → (f x) ‘R.≤‘ (f y);
end
```

Two examples of endomorphism of `NatPreorder` are:

⟨*successor as a* `NatPreorder` *morphism*⟩≡
```
struct
  D := NatPreorder;
  R := NatPreorder;
  f := Std.succ;
  compat := ⟨proof of compatibility of succ with ≤⟩;
end
```

⟨*zero as a* `NatPreorder` *morphism*⟩≡
```
struct
  D := NatPreorder;
  R := NatPreorder;
  f := λ x: Std.nat . Std.zero;
  compat := ⟨proof of compatibility of f with ≤⟩;
end
```

3.4 Parameterized modules

$\mathscr{MC}_2$ allows parameterized modules, called functors. These functors are useful for defining parameterized theories and for building mathematical structures.

Functors in $\mathscr{MC}_2$ are first-class modules, like functions are first-class values in programming languages. This means for instance that $\mathscr{MC}_2$ allows anonymous functors, higher-order functors and partial application of functors. This also means that a functor definition $x{:=}m{:}M$ is in no way treated differently from structure

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

definitions: we check the functor m implements the functor type M and add to the environment the binding $x : M$.[5]

Functor expressions are introduced by the keyword `functor` and functor type expressions by `funcT`.

3.4.1 Parameterized theories

A solution to the problem of handling parameterized theories is to use parameterized modules. Then, one can develop for instance a general theory `PreorderTheory` of preorders parameterized by a generic preorder `P`. This parameterized module can then be instantiated over `NatPreorder` or any other module implementing the ⟨*generic preorder signature*⟩.

The very basic theory we develop below defines the main definitions over preorders (upper bound, least upper bound, minimun, maximum, lub, glb) and shows that minimum, maximum, lub and glb are unique in the sense that if x and y are a couple of such elements then x and y are equivalent (*i.e.* $x \leqslant y$ and $y \leqslant x$).

⟨*preorder theory*⟩≡

```
functor P : ⟨generic preorder signature⟩ →
 struct
  a_subset := P.a → Prop;
  elt := λ x : P.a . λ s : a_subset . (s x);
  lower_bounds := λ s : a_subset . λ l : P.a .
    ∀ x : P.a . (x ‘elt‘ s) → (l ‘P.≤‘ x);
  lowest := λ s : a_subset . λ min : P.a .
    (min ‘elt‘ s) ‘Std.∧‘ (min ‘elt‘ (lower_bounds s));
  upper_bounds := λ s : a_subset . λ u : P.a .
    ∀ x : P.a . (x ‘elt‘ s) → (x ‘P.≤‘ u);
  greatest := λ s : a_subset . λ max : P.a .
    (max ‘elt‘ s) ‘Std.∧‘ (max ‘elt‘ (upper_bounds s));
  lub := λ s : a_subset . (lowest (upper_bounds s));
  glb := λ s : a_subset . (greatest (lower_bounds s));
  g_unicity := ⟨proof that for all x,y in (greatest s), x ⩽ y⟩;
  l_unicity := ⟨proof that for all x,y in (lowest s), x ⩽ y⟩;
  lub_unicity :=
    λ s : a_subset . (l_unicity (upper_bounds s));
  glb_unicity :=
    λ s : a_subset . (g_unicity (lower_bounds s));
 end
```

Instantiating this theory on `NatPreorder` is straightforward:

⟨*instantiation of* `PreorderTheory` *over* `NatPreorder`⟩≡

```
module NatPreorderTheory := (PreorderTheory NatPreorder)
```

[5] In other words, our functors have fully syntactic signatures.

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

Like for structures, giving an interface for a functor definition is optional. We could define PreorderTheory as follows

⟨PreorderTheory *definition*⟩≡
```
PreorderTheory := ⟨preorder theory⟩ : ⟨preorder theory interface⟩
```
where

⟨*preorder theory interface*⟩≡
```
funcT P : ⟨generic preorder signature⟩ →
 sig
  a_subset : Type := P.a → Prop;
  elt : P.a → a_subset → Prop :=
    λ x : P.a . λ s : a_subset . (s x);
  lower_bounds : a_subset → a_subset :=
    λ s : a_subset . λ l : P.a .
     ∀ x : P.a . (x ‘elt‘ s) → (l ‘P.⩽‘ x);
  lowest : a_subset → a_subset :=
    λ s : a_subset . λ min : P.a .
     (min ‘elt‘ s) ‘Std.∧‘ (min ‘elt‘ (lower_bounds s));
  upper_bounds : a_subset → a_subset :=
    λ s : a_subset . λ u : P.a .
     ∀ x : P.a . (x ‘elt‘ s) → (x ‘P.⩽‘ u);
  greatest : a_subset → a_subset :=
    λ s : a_subset . λ max : P.a .
     (max ‘elt‘ s) ‘Std.∧‘ (max ‘elt‘ (upper_bounds s));
  lub : a_subset → a_subset :=
    λ s : a_subset . (lowest (upper_bounds s));
  glb : a_subset → a_subset :=
    λ s : a_subset . (greatest (lower_bounds s));
  g_unicity : ∀ s : a_subset . ∀ x y : P.a .
    x ‘elt‘ (greatest s) →
      y ‘elt‘ (greatest s) → x ‘P.⩽‘ y;
  l_unicity : ∀ s : a_subset . ∀ x y : P.a .
    x ‘elt‘ (lowest s) →
      y ‘elt‘ (lowest s) → x ‘P.⩽‘ y;
  lub_unicity : ∀ s : a_subset . ∀ x y : P.a .
    x ‘elt‘ (lub s) → y ‘elt‘ (lub s) → x ‘P.⩽‘ y ;
  glb_unicity : ∀ s : a_subset . ∀ x y : P.a .
    x ‘elt‘ (glb s) → y ‘elt‘ (glb s) → x ‘P.⩽‘ y;
 end
```

3.4.2 *Building mathematical structures*

Another interesting use of functors is for building mathematical structures from other ones. For instance, one can build the opposite preorder OppositePreorder from a preorder, as follows:

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

⟨*opposite preorder*⟩≡
```
functor P: ⟨generic preorder signature⟩ →
 struct
  a := P.a;
  ⩽ := λ x y : a . y ‘P.⩽‘ x;
  refl := λ x : a . (P.refl x);
  trans := λ x y z : a . λ x_le_y : x ⩽ y .
    λ y_le_z : y ⩽ z . (P.trans z y x y_le_z x_le_y);
 end
```

In the same vein, we can define the product of preorders or the intersection preorder. Here is the definition of the intersection preorder `InterPreorder`:

⟨*intersection preorder*⟩≡
```
functor P1: ⟨generic preorder signature⟩ →
functor P2:
 sig
   a : Set := P1.a;
   ⩽ : a → a → Prop;
   refl : ∀x : a . x ⩽ x;
   trans : ∀x y z : a . x ⩽ y → y ⩽ z → x ⩽ z;
 end →
 struct
   a := P1.a;
   ⩽ := λ x y : a . (x ‘P1.⩽‘ y) ‘Std.∧‘ (x ‘P2.⩽‘ y);
   refl := ⟨proof of reflexivity of the intersection ⩽⟩;
   trans := ⟨proof of transitivity of the intersection ⩽⟩;
 end
```

`InterPreorder` can then be applied to any couple of preorders `P1` and `P2` such that `P1.a` and `P2.a` are convertible. For instance:

⟨*intersection of NatPreorder and its opposite*⟩≡
```
(InterPreorder NatPreorder (OppositePreorder NatPreorder))
```

Notice that manifest fields are essential here. Without them, for instance in theories with dependently typed records, one cannot build such a function. Instead, one has to find a workaround such as parameterizing preorder by their carrier: one gets a family of preorders indexed by their carrier and one can subsequently define a function taking as input a carrier A and two A-preorders, and returning an A-preorder.

3.5 Dealing with richer structures

3.5.1 Subtyping

The reader may have noticed that we could apply `PreorderTheory` to `NatPreorder` despite the ⟨*interface for NatPreorder*⟩ is slightly different from the ⟨*generic interface for preorder*⟩ that `PreorderTheory` expects as its argument.

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

More generally, at any place a structure of type M is expected in $\mathscr{MC}_2$, a richer structure may be put instead. In fact, $\mathscr{MC}_2$ is equipped with a subtyping relation $<:$ between module types and a typing rule saying that any module of type M_1 also has type M_2 provided M_1 is a subtype of M_2. It can be shown that $\mathscr{MC}_2$ enjoys principal types, that is, the set of types of a module expression has a smallest element with respect to the subtyping relation.

The informal intended meaning of $M_1 <: M_2$ is that the type M_1 is richer than or as rich as M_2. In other words, any module expression of type M_1 provides at least as much as an abstract module of type M_2. More precisely, if M_1 and M_2 are signatures, all fields of M_2 must be present in M_1 with a specification as demanding as or more demanding than the specification they have in M_2. This means that for providing a module of type M_2, one may provide a module m which has more fields than M_2 and whose fields are not in the same order as in M_2. Moreover, one may turn manifest fields of the principal type of m into abstract ones, like for the above application of `PreorderTheory` to `NatPreorder`.

This subtyping relation is extended to functor types with the usual contravariant rule: if M_1 and M_2 are two functor types, then M_1 is a subtype of M_2 if and only if the domain of M_2 is a subtype of M_1 and the range of M_1 is a subtype of the range of M_2.

3.5.2 Possible extensions

In addition to subtyping, programming languages have developed other ways to deal with enrichments of structures, which would be relevant in a module system for proofs.

Signature bindings and inheritance One can notice we used several times the module type ⟨*generic preorder signature*⟩. Such a repetition is in general a bad software engineering practice as it breaks the principle "do it once". As a consequence, more has to be typed by the user and each time a change is made to a development, it has to be done at several places.

SML-like module systems have a notion of module type variables and module type definitions to address this problem. Although such an addition seems harmless, it can have unsuspected effects: for instance adding *abstract* module type variables renders type-checking undecidable (Harper & Lillibridge, 1994). Therefore, we have not introduced such a feature in $\mathscr{MC}_2$ yet: although we doubt it would have endangered the logical consistency of $\mathscr{MC}_2$, it would certainly have complicated its metatheory a lot.

The usefulness of these module type abbreviations in SML is improved by two other constructs permitting signature inheritance: signature specialization and signature inclusion.

As for signature specialization, we could introduce a `where` construct in $\mathscr{MC}_2$, similar to the `where type` construct found in SML'97 (Milner *et al.*, 1997), called `with type` in Objective Caml (Leroy *et al.*, 2001). If `Preorder` is bound to ⟨*generic preorder signature*⟩ then one can write

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

⟨*example usage of the "where" construct*⟩≡
```
Preorder where a := Std.nat
```
in order to mean

⟨*meaning of* `Preorder where a := Std.nat`⟩≡
```
sig
 a : Set := Std.nat;
 ⩽ : a → a → Prop;
 refl : ∀x:a . x ⩽ x;
 trans : ∀x y z :a . x ⩽ y → y ⩽ z → x ⩽ z;
end
```

For signature inheritance, we could introduce an `include` construct for signature. For instance, the signature for a total preorder could be given as follows:

⟨*total preorder*⟩≡
```
sig
  include Preorder;
  total : ∀ x y : a . x ⩽ y ∨ y ⩽ x;
end
```

These features are undoubtly really useful when programming and would probably be as useful for making developments in abstract algebra. We identify them as two challenging areas for future work. Notice one difficulty is the notion of scope is now more subtle since the bindings of `Preorder` are now reopened in ⟨*total preorder*⟩.

Code inheritance Object-oriented languages propose also a way to deal with enrichment called code inheritance. Although some theoretical work try to address code inheritance for modules through mixins modules (Ancona & Zucca, 1996), it is not clear how to make them fit into SML-like module systems yet. However, a weaker form of code inheritance is already available in SML through the `open` construct (and in Objective Caml through the `include` construct). This construct roughly re-exports the body of a given module in the current structure. This is the kind of enrichment we want for mathematical structures: for instance, assume given a module `X` of type `Preorder` and assume we can prove the preorder on `X.a` is total. We would like to build a total preorder structure from `X` as follows:

⟨*building a total preorder from a given preorder*⟩≡
```
struct
  include X;
  total := ⟨proof of totality of X.⩽⟩;
end
```
with the meaning:

⟨*meaning of the preceding*⟩≡
```
struct
  a := X.a;
  ⩽ := X.‘⩽‘;
  refl := X.refl;
```

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

```
  trans := X.trans;
  total := ⟨proof of totality of X.≤⟩;
end
```

Undoubtedly, using this include construct would let us define structures in a more modular fashion. Unfortunately, this include construct makes subject-reduction property fail in presence of subtyping. To see this, consider the following structure:

```
⟨counter-example to substitution property in presence of include⟩≡
  struct
    a : Set;
    include X;
    b : a;
  end
```

Consider an environment declaring `Y: sig a : Std.nat; end` and `X : sig end`. The above structure is well-typed but if X is substituted by Y, which is legal thanks to subtyping, it becomes ill-typed. In other words, the substitution property does not hold: not only subject-reduction fails but the upward compatibility property (introduced in Section 1.1.1 and formally defined in Section 5.2) also fails.

A possibly safe way to add such a construct would be to require the include construct to be given with a list of re-exported fields. The include construct would thus become an "`include` x, y, ... `from` m", similar to Modula 2 "`FROM` m `IMPORT` x, y, ..." construct. Another way would be to require the include to be given together with a signature, either an explicit one — "`include sig` x : ... ; y : ... ; ... `end from` m" would have the meaning of the previous "`include` x, y, ... `from` m" — or through signature abbreviation — "`include` S `from` x" would have the meaning of "`include` S' `from` m" where S' is the normal form of S. In both cases, ensuring the soundness of the extended calculus would be done by giving a translation of the extended calculus into $\mathscr{MC}_2$. The metatheory of such an extended calculus would essentially amount to proving the compatibility of this translation with reduction. As for the former construct, it would be quite trivial as the translation to $\mathscr{MC}_2$ is trivial (it can be defined as a straightforward, local, rewriting, clearly commuting with reduction). As for the latter, the proof would certainly be more difficult as the translation to $\mathscr{MC}_2$ might require the normalization of the given abbreviation first, which could leave room for unexpected interactions with reduction at other places.

3.6 Transparent module definitions

We said earlier that, after it has processed a module definition x`:=`m, the type-checker forgets m. As a consequence, x and m are then considered different by the type checker.

This might be annoying in some rare circumstances. Therefore, $\mathscr{MC}_2$ has a special notion of module definition, called *transparent module definition*: after a definition x`:=transparent` m, the type-checker retains the equality $x = m$.

We give here an example of such a circumstance: preorder morphism composition. We can define a functor taking as input two preorder morphisms and composing

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

them. This functor would have the following type:

⟨*preorder morphism composition functor type*⟩≡
```
funcT P1: ⟨generic preorder morphism signature⟩ →
funcT P2:
  sig
    D: ⟨generic preorder signature⟩ := P1.R;
    R: ⟨generic preorder signature⟩;
    f: D.a → R.a;
    compat: ∀x y : D.a . x ‘D.⩽‘ y → (f x) ‘R.⩽‘ (f y);
  end →
  sig
   D: ⟨generic preorder signature⟩ := P1.D;
   R: ⟨generic preorder signature⟩ := P2.R;
   f: D.a → R.a := λ x: D.a . (P2.f (P1.f x));
   compat: ∀x y : D.a . x ‘D.⩽‘ y → (f x) ‘R.⩽‘ (f y);
  end
```

If we try to apply this functor to the morphisms `SuccPreorder` and `ZeroPreorder` defined in section 3.3, the type-checker has to check that `SuccPreorder.R` and `ZeroPreorder.D` are equal. This test fails although both of them are defined as being `NatPreorder`. Indeed, once a module definition has been given, its contents is thrown away and only its interface is kept by the type-checker. Therefore, the type-checker cannot reduce `SuccPreorder.R` nor `ZeroPreorder.D`: they are distinct normal forms.

In order to circumvent this problem, the fields `D` and `R` of a preorder morphism should be defined as transparent modules:

⟨*successor as a* `NatPreorder` *morphism using transparent modules*⟩≡
```
SuccPreorder :=
 struct
  D := transparent NatPreorder;
  R := transparent NatPreorder;
  f := succ;
  compat := ⟨proof of compatibility of succ with ⩽⟩;
 end
```

With this definition, the fields `D` and `R` of the interface of `SuccPreorder` are manifest:

⟨*inferred interface for* `SuccPreorder` *with transparent modules*⟩≡
```
sig
 D : ⟨NatPreorder interface⟩ := NatPreorder;
 R : ⟨NatPreorder interface⟩ := NatPreorder;
 f : D.a → R.a := succ;
 compat: ∀x y : D.a . x ‘D.⩽‘ y → (f x) ‘R.⩽‘ (f y)
  := ⟨proof of compatibility of succ with ⩽⟩;
end
```

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

Whereas in the inferred interface for the definition of `SuccPreorder` given in section 3.3, the constraints `D := NatPreorder` and `R := NatPreorder` do not appear:

⟨*inferred interface for* `SuccPreorder` *without transparent modules*⟩≡

```
sig
 D : ⟨NatPreorder interface⟩;
 R : ⟨NatPreorder interface⟩;
 f : D.a → R.a := succ;
 compat: ∀x y : D.a . x ‘D.⩽‘ y → (f x) ‘R.⩽‘ (f y)
   := ⟨proof of compatibility of succ with ⩽⟩;
end
```

3.7 Separate checking

In practice, a proof development in **oeuf** is as a list of pairs of files

$$(x_1\texttt{.iv}, x_1\texttt{.pv}), \ldots, (x_n\texttt{.iv}, x_n\texttt{.pv})$$

Each file x_i`.iv` is an *interface vernacular* file containing a module type M_i and each file x_i`.pv` is a *proof vernacular* file containing a module expression m_i. The meaning of this proof development is the module expression

```
struct
  module x_1 : M_1 = m_1;
  ⋮
  module x_n : M_n = m_n;
end
```

This development can be separately checked by **oeuf**, which means that a given proof vernacular file x_i`.pv` can be checked in absence of other proof vernacular files: checking m_i can be done with the sole knowledge of the interface vernacular files it depends on.

For instance, the examples given above in this section could be split accross several files as follows.

First of all, we put in interface files the interfaces of the modules we want to develop and we check them in an order compatible with their dependencies:

⟨*Std.iv*⟩≡

```
sig
(* Standard declarations: and, or, natural numbers, ... *)
  ⟨standard declarations⟩
end
```

⟨*NatPreorder.iv*⟩≡

```
  ⟨NatPreorder interface⟩
```

⟨*ZeroPreorder.iv*⟩≡

```
sig
 D : ⟨NatPreorder interface⟩;
```

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

```
  R : ⟨NatPreorder interface⟩;
  f : Std.nat → Std.nat;
  compat : ∀x y : D.a . x `D.⩽` y→ (f x) `R.⩽` (f y);
 end
```

⟨*SuccPreorder.iv*⟩≡

```
 sig
  D : ⟨NatPreorder interface⟩;
  R : ⟨NatPreorder interface⟩;
  f : Std.nat → Std.nat;
  compat : ∀x y : D.a . x `D.⩽` y→ (f x) `R.⩽` (f y);
 end
```

⟨*NatPreorderTheory.iv*⟩≡

⟨*preorder theory interface*⟩

⟨*OppositePreorder.iv*⟩≡

```
 funcT P : ⟨generic preorder signature⟩ →
  sig
   a : Set := P.a;
   ⩽ : a → a → Prop;
   refl : ∀x:a . x ⩽ x;
   trans : ∀x:a . ∀y:a . ∀z:a . x ⩽ y → y ⩽ z → x ⩽ z;
  end
```

⟨*InterPreorder.iv*⟩≡

```
  funcT P1: ⟨generic preorder signature⟩ →
  funcT P2: sig
             a : Set := P1.a;
             ⩽ : a → a → Prop;
             refl : ∀x : a . x ⩽ x;
             trans : ∀x y z : a . x ⩽ y → y ⩽ z → x ⩽ z;
            end →
 sig
  a : Set := P1.a;
  ⩽ : a → a → Prop;
  refl : ∀x:a . x ⩽ x;
  trans : ∀x:a . ∀y:a . ∀z:a . x ⩽ y → y ⩽ z → x ⩽ z;
 end
```

⟨*Main.iv*⟩≡

⟨*generic preorder signature*⟩

Then, the following implementation files can be checked in any order:

⟨*NatPreorder.pv*⟩≡

⟨*structure of preorder over natural numbers*⟩

⟨*ZeroPreorder.pv*⟩≡

⟨*zero as a* `NatPreorder` *morphism*⟩

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

⟨*SuccPreorder.pv*⟩≡
⟨*successor as a* `NatPreorder` *morphism*⟩

⟨*NatPreorderTheory.pv*⟩≡
⟨*preorder theory*⟩

⟨*OppositePreorder.pv*⟩≡
⟨*opposite preorder*⟩

⟨*InterPreorder.pv*⟩≡
⟨*intersection preorder*⟩

⟨*Main.pv*⟩≡
⟨*intersection of NatPreorder and its opposite*⟩

Then our metatheoretical results ensure the whole development is correct. As a consequence, if one modifies an implementation file, only this file has to be checked again.

4 Formal presentation

We can now give a more formal presentation of $\mathcal{MC}_2$. As $\mathcal{MC}_2$ is independent of the chosen PTS, we fix an arbitrary choice of a PTS, that is, a set of sorts $\mathcal{S}$, a set of axioms $\mathcal{A}$ and a set of products $\mathcal{R}$. And we note $\mathcal{MC}_2(\mathcal{S},\mathcal{A},\mathcal{R})$ the associated module extension of this PTS.

We first describe the grammar and the associated rules of $\mathcal{MC}_2(\mathcal{S},\mathcal{A},\mathcal{R})$ in section 4.2. Then in section 4.3 we define the convertibility notion we use in $\mathcal{MC}_2(\mathcal{S},\mathcal{A},\mathcal{R})$. Finally, we describe the rules governing subtyping and type conversion rules in section 4.4.

4.1 Conventions used in this article

Metavariables names are meaningful In all this paper, we make the convention that *the metavariable names we use for non-terminals in the grammar given figure 1 are meaningful*. For instance metavariables such as $m, m', m_1, \ldots$ range over module expressions, and $M, M', M_1, \ldots$ range over module types. Thus, if we state that "for all m ... ", we actually mean that "for all module expression m ... ".

Moreover, we use q to denote an expression that can be either a base term or a module expression, τ to denote an expression that can be either a base term or a module type, and γ to denote any kind of expression.

Renaming For the sake of clarity, we deliberately ignore renaming problems in the following. We refer the interested reader to appendix C to see how we deal with this issue. For the first reading, the reader may simply consider that a variable can never be bound twice in a given environment.

4.2 Syntax and syntax-directed rules

$\mathcal{MC}_2(\mathcal{S},\mathcal{A},\mathcal{R})$ involves syntactic categories and several different judgments. We gave each inference rule of $\mathcal{MC}_2(\mathcal{S},\mathcal{A},\mathcal{R})$ a name of the form *foo*/*bar*, where *foo*

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

Types

Module types

M ::=	`sig` S `end`	signature
|	`funcT` $x : M \to M$	functor type

Signature bodies

S ::=	ϵ	empty body
|	w : $\mathfrak{S}$; S	

Specifications

$\mathfrak{S}$ ::=	M	abstract module
|	M := m	manifest module
|	t	abstract term
|	t := t	manifest term

Environments

Γ ::=		empty environment
|	Γ ; x : $\mathfrak{S}$	adding a binding

Terms

Module expressions

m ::=	`struct` s `end`	structure
|	`functor` $x : M \to m$	functor
|	(m m)	application
|	p	module path

Paths

p ::=	x	variable
|	m . x	field selection

Structure bodies

s ::=	ϵ	empty body
|	x := d ; s	

Definitions

d ::=	t	term definition
|	m	module definition
|	m : M	module definition with interface
|	`transparent` m	transparent module definition

Base terms

t ::=	$\lambda\, x : t$. t	abstraction
|	$\forall x : t$. t	product
|	(t t)	application
|	σ	sort
|	p	term path

Fig. 1. Grammar of $\mathscr{MC}_2(\mathscr{S}, \mathscr{A}, \mathscr{R})$

determines uniquely the kind of judgment involved in the conclusion of the rule. For instance, T for typing judgments for base terms, M for typing judgment for module expressions, SUB for subtyping between module types... In figure 1, we give the grammar of $\mathscr{MC}_2(\mathscr{S}, \mathscr{A}, \mathscr{R})$ and in figure 2, we give the list of judgments $\mathscr{MC}_2(\mathscr{S}, \mathscr{A}, \mathscr{R})$ involves together with their informal meaning and their associated prefix.

We can now explain the syntax of $\mathscr{MC}_2(\mathscr{S}, \mathscr{A}, \mathscr{R})$ by giving the associated inference rules.

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

Well-formedness judgments

[MT/] $\Gamma \vdash M$: modtype "M is a well-formed module type in the environment Γ";
[SB/] $\Gamma \vdash S$: sigbody "S is a well-formed signature body in Γ".
[WFS/] $\Gamma \vdash \mathfrak{S}$: wfs "$\mathfrak{S}$ is a well-formed specification in Γ";
[ENV/] $\Gamma \vdash$ ok "Γ is a well-formed environment" ;

Typing judgments

[M/] $\Gamma \vdash m : M$ "the module expression m has type M in Γ";
[S/] $\Gamma \vdash s :: S$ "the structure body s has type the signature body S in Γ";
[DEF/] $\Gamma \vdash d :: \mathfrak{S}$ "the definition d introduces an expression of specification $\mathfrak{S}$ in Γ";
[SPEC/] $\Gamma \vdash p : \mathfrak{S}$ "the path p has specification $\mathfrak{S}$ in Γ";
[T/] $\Gamma \vdash t : t'$ "the base term t has type t' in Γ".

Subtyping judgment

[SUB/] $\Gamma \vdash M_1 <: M_2$ "M_1 is a subtype of M_2 in Γ";
[SUBSPEC/] $\Gamma \vdash \mathfrak{S}_1 <: \mathfrak{S}_2$ "$\mathfrak{S}_1$ is a sub-specification of $\mathfrak{S}_2$ in Γ";
[ENT/] $\Gamma|_{dom} \vdash S$ "Γ restricted to *dom* entails S".

Fig. 2. Judgments of $\mathscr{MC}_2(\mathscr{S}, \mathscr{A}, \mathscr{R})$

4.2.1 Well-formedness conditions for types

Module types A *module type* is either a signature containing a list of declarations called a *signature body* or a functor type.

As for signatures, we define Alldiff(S) as the proposition "All the names of the fields of S are pairwise distinct". A signature is well-formed as soon as its body S is well-formed and Alldiff(S) holds:

$$\mathsf{MT/SIG}\frac{\Gamma \vdash S : \mathsf{sigbody} \qquad \mathrm{Alldiff}(S)}{\Gamma \vdash \mathtt{sig}\ S\ \mathtt{end} : \mathsf{modtype}}$$

A functor type is well-formed if its domain and its range are well-formed:

$$\mathsf{MT/PROD}\frac{\Gamma \vdash M : \mathsf{modtype} \qquad \Gamma; x : M \vdash M' : \mathsf{modtype}}{\Gamma \vdash \mathtt{funcT}\ x{:}M \to M' : \mathsf{modtype}}$$

Signature bodies A signature body is a list of declarations, each declaration being a pair formed of a field name and the specification of this field. A signature body is well-formed if and only if it contains only well-formed specifications.

Thus, the empty body is well-formed:

$$\mathsf{SB/EMPTY}\frac{}{\Gamma \vdash \epsilon : \mathsf{sigbody}}$$

And a non-empty one is well-formed if its first declaration has a well-formed specification and the remaining signature body is well-formed:

$$\mathsf{SB/DECL}\frac{\Gamma \vdash \mathfrak{S} : \mathsf{wfs} \qquad \Gamma; x : \mathfrak{S} \vdash S : \mathsf{sigbody}}{\Gamma \vdash x{:}\mathfrak{S}; S : \mathsf{sigbody}}$$

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

Specifications A field declaration in a signature body can either simply reveal a type (in which case the field is given an *abstract specification*) or it can reveal a type and the value of the declared field (in which case the field is given a *manifest specification*). A field declaration is either a module specification or a term specification. Hence, there are four different kinds of specifications; each one has an associated rule.

If the specification is abstract, checking it is well-formed just requires to check that it contains a type (*i.e.* a module type or a base term whose type is a sort):

$$\mathsf{WFS/ABSTERM}\frac{\Gamma \vdash t : \sigma}{\Gamma \vdash t : \mathsf{wfs}}$$

$$\mathsf{WFS/ABSMOD}\frac{\Gamma \vdash M : \mathsf{modtype}}{\Gamma \vdash M : \mathsf{wfs}}$$

If the specification is manifest, it requires to check that its manifest part inhabits its type part:

$$\mathsf{WFS/MANTERM}\frac{\Gamma \vdash t : t'}{\Gamma \vdash t' :\!= t : \mathsf{wfs}}$$

$$\mathsf{WFS/MANMOD}\frac{\Gamma \vdash m : M}{\Gamma \vdash M :\!= m : \mathsf{wfs}}$$

4.2.2 *Well-formed environments*

An environment is a list of bindings from variables to specifications. Variables declared with an abstract specification are similar to variables in type theory. Variables declared with a manifest specification $\tau :\!= q$ are similar to definitions such as the ones found in (Severi, 1996).

An environment is said "well-formed" if and only if all the specifications it contains are well-formed:

$$\mathsf{ENV/EMPTY}\frac{}{\vdash \mathsf{ok}}$$

$$\mathsf{ENV/DECL}\frac{\Gamma \vdash \mathsf{ok} \qquad \Gamma \vdash \mathfrak{S} : \mathsf{wfs}}{\Gamma; x : \mathfrak{S} \vdash \mathsf{ok}}$$

4.2.3 *Typing judgments*

Module expressions A module expression is either a structure, a functor, an application or a path.

Typing a structure just amounts to typing the declarations it contains and checking they have pairwise distinct names:

$$\mathsf{M/STRUCT}\frac{\Gamma \vdash s :: S \qquad \mathrm{Alldiff}(S)}{\Gamma \vdash \mathtt{struct}\ s\ \mathtt{end} : \mathtt{sig}\ S\ \mathtt{end}}$$

Handling parameterized modules just requires a rule for application and one for abstraction. These rules are the standard rules for application and abstraction in systems with dependent types.

$$\mathsf{M/APP}\frac{\Gamma \vdash m_1 : \mathtt{funcT}\ x{:}M \to M' \qquad \Gamma \vdash m_2 : M}{\Gamma \vdash (m_1\ m_2) : M'\{x \leftarrow m_2\}}$$

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

$$\text{M/LAM}\frac{\Gamma \vdash M : \texttt{modtype} \qquad \Gamma; x : M \vdash m : M'}{\Gamma \vdash \texttt{functor}\ x{:}M \to m : \texttt{funcT}\ x{:}M \to M'}$$

In order to get the type of a module path, we compute its specification and just take the type if contains:

$$\text{M/PATH}\frac{\Gamma \vdash p : \mathfrak{S}}{\Gamma \vdash p : ty(\mathfrak{S})}$$

where $ty(\mathfrak{S})$ denotes the type component of the specification $\mathfrak{S}$. More formally, for all τ and q, we define

$$\begin{aligned} ty(\tau\texttt{:=}q) &= \tau \\ ty(\tau) &= \tau \end{aligned}$$

Paths Paths are either a variable or a field selection.

The specification of a variable just has to be looked up into the environment:

$$\text{SPEC/VAR}\frac{}{\Gamma \vdash x : \Gamma(x)}$$

As for field selections, $m.x$ is well-typed if and only if m has type `sig` S `end` and x is the name of a field declared in S. Then the specification of $m.x$ is the specification $\mathfrak{S}$ associated to x in S where for every field x' preceding x, x' was substituted by $m.x'$.

More formally, we define field(m, x, S) as the specification $\mathfrak{S}$ associated to x in S where for every field x' preceding x, x' was substituted by $m.x'$:

$$\begin{aligned} \text{field}: \quad (m, x, (x{:}\mathfrak{S}\,;S)) &\mapsto \mathfrak{S} \\ (m, x, (x'{:}\mathfrak{S}\,;S)) &\mapsto \text{field}(m, x, S\{x' \leftarrow m.x'\}) \text{ if } x \neq x' \end{aligned}$$

Then the rule for typing a field selection is:

$$\text{SPEC/SELECT}\frac{\Gamma \vdash m : \texttt{sig}\ S\ \texttt{end} \qquad \text{field}(m, x, S) = \mathfrak{S}}{\Gamma \vdash m.x : \mathfrak{S}}$$

Structure bodies A structure body is a list of field definitions; the type of a structure body is the list of the respective specifications of its fields.

The type of the empty structure body is the empty signature body:

$$\text{S/EMPTY}\frac{}{\Gamma \vdash \epsilon :: \epsilon}$$

In order to type a non-empty structure body, we first infer the specification $\mathfrak{S}$ of the first field's definition d, then we bind the name of this field to $\mathfrak{S}$ in the environment and infer the signature body of the remaining structure body in this new environment:

$$\text{S/DECL}\frac{\Gamma \vdash d :: \mathfrak{S} \qquad \Gamma\ ;\ x : \mathfrak{S} \vdash s :: S}{\Gamma \vdash x{:=}d\,;s :: x{:}\mathfrak{S}\,;S}$$

Definitions A definition can be either a term definition or a module definition.

A term definition t_1 is well-typed as soon as t_1 is well-typed. Then, the specification of the given definition is the manifest term specification whose type part is the type

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

of t_1 and whose manifest part is t_1:

$$\mathsf{DEF/TERM}\frac{\Gamma \vdash t_1 : t_2}{\Gamma \vdash t_1 :: t_2 := t_1}$$

As for module definitions, a specification of a module with no explicit interface is just any abstract specification whose type part is a type of the module:

$$\mathsf{DEF/M1}\frac{\Gamma \vdash m : M}{\Gamma \vdash m :: M}$$

The specification of a module m given with an explicit interface M is the abstract specification M provided M is a type of m:

$$\mathsf{DEF/M2}\frac{\Gamma \vdash m : M}{\Gamma \vdash m : M :: M}$$

Finally, a transparent module definition has any specification whose type part is any type of the module and whose manifest part is the module:

$$\mathsf{DEF/TRANSPM}\frac{\Gamma \vdash m : M}{\Gamma \vdash \texttt{transparent}\ m :: M := m}$$

Base terms The rules for base terms are the standard PTS rules, with two exceptions:

- The rule for variables has been replaced by a rule for paths identical to that for modules paths.
- The conversion rule does not relies on β nor $\beta\eta$ equality but on a convertibility relation $\bowtie$, defined in section 4.3 below.

$$\mathsf{T/LAM}\frac{\Gamma \vdash t_1 : \sigma_1 \qquad \Gamma; x : t_1 \vdash t_2 : t_3 \qquad \Gamma \vdash \forall x : t_1 . t_3 : \sigma_2}{\Gamma \vdash \lambda x : t_1 . t_2 : \forall x : t_1 . t_3}$$

$$\mathsf{T/PROD}\frac{\Gamma \vdash t_1 : \sigma_1 \qquad \Gamma; x : t_1 \vdash t_2 : \sigma_2 \qquad (\sigma_1, \sigma_2, \sigma_3) \in \mathscr{R}}{\Gamma \vdash \forall x : t_1 . t_2 : \sigma_3}$$

$$\mathsf{T/APP}\frac{\Gamma \vdash t_1 : \forall x : t_4 . t_5 \qquad \Gamma \vdash t_2 : t_4}{\Gamma \vdash (t_1\ t_2) : t_5\{x \leftarrow t_2\}}$$

$$\mathsf{T/SORT}\frac{(\sigma_1, \sigma_2) \in \mathscr{A}}{\Gamma \vdash \sigma_1 : \sigma_2}$$

$$\mathsf{T/PATH}\frac{\Gamma \vdash p : \mathfrak{S}}{\Gamma \vdash p : ty(\mathfrak{S})}$$

$$\mathsf{T/CONV}\frac{\Gamma \vdash t_1 : t_3 \qquad \Gamma \vdash t_2 : \sigma \qquad \Gamma \vdash t_2 \bowtie t_3}{\Gamma \vdash t_1 : t_2}$$

4.3 Convertibility

One of the usual PTS typing rules is the conversion rule, involving convertibility of terms. In $\mathscr{MC}_2$, the conversion test involves relations tied to module declarations in addition to the usual β-reduction. We introduce here the reductions we consider. In this purpose, we first introduce the generic definitions and notations we use.

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

Definition 4.1 (*Env-dependent relation*)
An *env-dependent relation* R is a relation over triples $(\Gamma, \gamma, \gamma')$, where Γ is an environment and γ and γ' are expressions. If R is an env-dependent relation, we note $\Gamma \vdash \gamma R \gamma'$ to mean $R(\Gamma, \gamma', \gamma')$.

Definition 4.2 (*R-redex*)
If R is an env-dependent relation, we say γ is a R-redex in Γ if and only if there exists γ' such that $\Gamma \vdash \gamma R \gamma'$. In that case, we say γ' is the result of the contraction of the R-redex γ.

Definition 4.3 (*Context*)
A *context* C is an expression containing a single occurrence of the variable $\circ$. We note $C[\gamma]$ the expression $C\{\circ \leftarrow \gamma\}$. We note $\mathscr{P}(C)$ the list of bindings crossed in C in order to reach $\circ$. For instance, we have

$$\begin{array}{rcll}
\mathscr{P}(\circ) & = & \epsilon & \\
\mathscr{P}(\texttt{funcT}\ x{:}M \to M') & = & x : M; \mathscr{P}(M') & \text{if } \circ \text{ appears in } M' \\
\mathscr{P}(\texttt{funcT}\ x{:}M \to M') & = & \mathscr{P}(M) & \text{if } \circ \text{ appears in } M \\
\mathscr{P}(\texttt{functor}\ x{:}M \to m) & = & x : M; \mathscr{P}(m) & \text{if } \circ \text{ appears in } m \\
\mathscr{P}(\texttt{functor}\ x{:}M \to m) & = & \mathscr{P}(M) & \text{if } \circ \text{ appears in } M \\
\mathscr{P}(x{:}\mathfrak{S}; S) & = & x : \mathfrak{S}; \mathscr{P}(S) & \text{if } \circ \text{ appears in } S \\
\mathscr{P}(x{:}\mathfrak{S}; S) & = & \mathscr{P}(\mathfrak{S}) & \text{if } \circ \text{ appears in } \mathfrak{S} \\
\mathscr{P}((m_1\ m_2)) & = & \mathscr{P}(m_1) & \text{if } \circ \text{ appears in } m_1 \\
\mathscr{P}((m_1\ m_2)) & = & \mathscr{P}(m_2) & \text{if } \circ \text{ appears in } m_2 \\
\mathscr{P}(\lambda x{:}t_1 . t_2) & = & x : t_1; \mathscr{P}(t_2) & \text{if } \circ \text{ appears in } t_2 \\
\mathscr{P}(\lambda x{:}t_1 . t_2) & = & \mathscr{P}(t_1) & \text{if } \circ \text{ appears in } t_1 \\
\mathscr{P}(\forall x{:}t_1 . t_2) & = & x : t_1; \mathscr{P}(t_2) & \text{if } \circ \text{ appears in } t_2 \\
\mathscr{P}(\forall x{:}t_1 . t_2) & = & \mathscr{P}(t_1) & \text{if } \circ \text{ appears in } t_1
\end{array}$$

Definition 4.4 (*Monotonic relation*)
Let R be an env-dependent relation. We say R is *monotonic* if and only if, for all context C, all environment Γ and all expressions γ and γ' such that $R(\Gamma; \mathscr{P}(C), \gamma, \gamma')$ we have $R(\Gamma, C[\gamma], C[\gamma'])$.

Remark 4.5
Let R be an env-dependent relation. If for some binary relation R', we have for all Γ, γ and γ', $R(\Gamma, \gamma, \gamma') = R'(\gamma, \gamma')$ (that is, if R is constant with respect to the environment) then R is monotonic in the sense given below if and only if R' is monotonic in the usual sense ("for all context C and all γ and γ' such that $R'(\gamma, \gamma')$, we have $R'(C[\gamma], C[\gamma'])$").

Definition 4.6 (*One step R-reduction*)
Let R be an env-dependent relation. We call one-step R-reduction the least monotonic relation containing R. We note this relation $\triangleright_R$. If $\Gamma \vdash \gamma \triangleright_R \gamma'$, we say γ R-reduces to γ' in one step in the environment Γ.

Definition 4.7 (*Transitive relation*)
Let R be an env-dependent relation. We say R is *transitive* if and only if, for all environment Γ and all expressions γ, γ' γ'' such that $\Gamma \vdash \gamma R \gamma'$ and $\Gamma \vdash \gamma' R \gamma''$, we

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

have $\Gamma \vdash \gamma R \gamma''$. Given an env-dependent relation R, the least transitive env-dependent relation containing it is called the *transitive closure* of R, and is noted $R^\star$.

Definition 4.8 (*Many steps R-reduction*)
Let R be an env-dependent relation. We call many-step R-reduction the relation $\triangleright_R^\star$. We note this relation $\triangleright_R^\star$. We say γ R-reduces to γ' in many steps in the environment Γ if $\Gamma \vdash \gamma \triangleright_R^\star \gamma'$.

Notation 4.9
If $R_1, R_2, \ldots, R_n$ are n relations over triples $(\Gamma, \gamma, \gamma')$, we note $R_1 R_2 \ldots R_n$ their union.

4.3.1 β-reduction

Definition 4.10 (*The β relation*)
β is the least env-dependent relation such that for all base terms t_1, t_2 and t_3, for all variable x, $\Gamma \vdash (\lambda x{:}t_1 . t_2\ t_3)\ \beta\ t_2\ \{x \leftarrow t_3\}$.

4.3.2 β_m-reduction

Functors also introduce a kind of β-redexes at the level of module expressions, called β_m-redexes:

Definition 4.11 (*The β_m relation*)
β_m is the least env-dependent relation such that for all module expressions m_1 and m_2, all module type M and all variable x, $\Gamma \vdash (\texttt{(functor}\ x{:}M \to m_1)\ m_2)\ \beta_m\ m_1\{x \leftarrow m_2\}$.

4.3.3 ρ-reduction

The selection of a field of a structure is a new kind of redex that can be reduced. First, we must define the notion of content of a definition:

Definition 4.12 (*Content of a definition*)
Let d be a definition. The content of d, noted $\mathscr{C}(d)$ is defined as follows:

$$\begin{aligned} \mathscr{C}(t) &= t \\ \mathscr{C}(m) &= m \\ \mathscr{C}(m{:}M) &= m \\ \mathscr{C}(\texttt{transparent}\ m) &= m \end{aligned}$$

We are now ready to define the ρ relation.

Definition 4.13 (*The ρ relation*)
ρ is the least env-dependent relation such that for all positive integers j and n, with $j \leqslant n$, for all sequence $x_1, \ldots, x_n$ of variables, for all sequence $d_1, \ldots, d_n$ of definitions we have

$$\Gamma \vdash m.x_j\ \rho\ \mathscr{C}(d_j)\{x_i \leftarrow m.x_i | i \in [1,n]\}$$

where m denotes $\texttt{struct}\ x_1{:=}d_1\texttt{;}\ \ldots\ \texttt{;}x_n{:=}d_n\texttt{;}\ \texttt{end}$

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

4.3.4 δ-reduction

As we have seen section 3.2.1, the type-checker sometimes needs to reduce an expression such as `NatPreorder.a` to `nat`, where `NatPreorder` is a *variable*. This means in addition to $\beta\beta_m\rho$-reduction, we need a new reduction, we call δ-reduction whose purpose is to reduce manifest field to their declared value. The idea behind the definition of δ-reduction is that $m.x$ is a redex if the field x appears as manifest in the type of m.

We used conversion (hence δ-reduction) to define the type system of $\mathcal{MC}_2$ and we are now saying that in order to δ-reduce an expression we need to know its type. So it might seems we have a chicken and egg problem here. Fortunately, we do not. In fact, we introduce in this section a *pseudo-typing* notion whose definition does not rely on equality. Then $m.x$ is a δ-redex if x appears as manifest in the pseudo-type of m. We introduce four judgments whose informal aim is to compute the pseudo-type of an expression:

- $\Gamma \vdash m \mathfrak{D} M$ "the module m has pseudo-type M"
- $\Gamma \vdash p \mathfrak{D} \mathfrak{S}$ "the path p has pseudo-specification $\mathfrak{S}$"
- $\Gamma \vdash s \mathfrak{D}\mathfrak{D} S$ "the structure body s has pseudo-signature body S"
- $\Gamma \vdash d \mathfrak{D}\mathfrak{D} \mathfrak{S}$ "the definition d has pseudo-specification $\mathfrak{S}$".

Definition 4.14 (*The δ relation*)
We define the δ env-dependent relation as the least relation such that for all path p and all environment Γ, $\Gamma \vdash p\ \delta q$ holds if and only if we can derive $\Gamma \vdash p \mathfrak{D}\ \tau\texttt{:=}q$ for some τ under the rules given figure 3.

Remark 4.15
The δ-rules are similar to the inference rules for typing modules expressions, paths, structure bodies and definitions given previously. For all but two of them ([Δ/DEF/TRANSPM] and [Δ/DEF/TERM]), the difference is that some premises have been omitted.

As for rules [Δ/DEF/TRANSPM] and [Δ/DEF/TERM], we gave both of them a manifest specification. In order not to compute the type of m nor of t_1 the specifications we give for them have fake type components (`sig` ϵ `end` and t_1). This is not a problem since we are only interested in the expression they reduce to.

4.3.5 Convertibility

Definition 4.16 ($\triangleright$)
We note $\triangleright$ the relation $\triangleright_{\beta\beta_m\rho\delta}$.

Definition 4.17 (*Convertibility*)
We say two expressions γ_1 and γ_2 are convertible in an environment Γ and we write $\Gamma \vdash \gamma_1 \bowtie \gamma_2$ if and only if there exists γ such that $\Gamma \vdash \gamma_1 \triangleright^\star \gamma$ and $\Gamma \vdash \gamma_2 \triangleright^\star \gamma$.

Remark 4.18
We do *not* define convertibility as the reflexive symmetric transitive closure of $\triangleright$. Indeed, $\triangleright^\star$ is not Church-Rosser on untyped expressions and its reflexive symmetric

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

Module expressions

$$\Delta/\mathsf{M}/\mathsf{APP}\frac{\Gamma \vdash m_1 \ \mathfrak{D}\ \texttt{funcT}\ x{:}M \to M'}{\Gamma \vdash (m_1\ m_2)\ \mathfrak{D}\ M'\{x \leftarrow m_2\}}$$

$$\Delta/\mathsf{M}/\mathsf{LAM}\frac{\Gamma; x : M \vdash m\ \mathfrak{D}\ M'}{\Gamma \vdash \texttt{functor}\ x{:}M \to m\ \mathfrak{D}\ \texttt{funcT}\ x{:}M \to M'}$$

$$\Delta/\mathsf{M}/\mathsf{STRUCT}\frac{\Gamma \vdash s\ \mathfrak{DD}\ S}{\Gamma \vdash \texttt{struct}\ s\ \texttt{end}\ \mathfrak{D}\ \texttt{sig}\ S\ \texttt{end}}$$

$$\Delta/\mathsf{M}/\mathsf{PATH}\frac{\Gamma \vdash p\ \mathfrak{D}\ \mathfrak{S}}{\Gamma \vdash p\ \mathfrak{D}\ ty(\mathfrak{S})}$$

Definitions

$$\Delta/\mathsf{DEF}/\mathsf{M1}\frac{\Gamma \vdash m\ \mathfrak{D}\ M}{\Gamma \vdash m\ \mathfrak{DD}\ M}$$

$$\Delta/\mathsf{DEF}/\mathsf{M2}\frac{}{\Gamma \vdash m{:}M\ \mathfrak{DD}\ M}$$

$$\Delta/\mathsf{DEF}/\mathsf{TRANSPM}\frac{}{\Gamma \vdash \texttt{transparent}\ m\ \mathfrak{DD}\ \texttt{sig}\ \epsilon\ \texttt{end}{:=}m}$$

$$\Delta/\mathsf{DEF}/\mathsf{TERM}\frac{}{\Gamma \vdash t_1\ \mathfrak{DD}\ t_1{:=}t_1}$$

Paths

$$\Delta/\mathsf{SPEC}/\mathsf{VAR}\frac{}{\Gamma \vdash x\ \mathfrak{D}\ \Gamma(x)}$$

$$\Delta/\mathsf{SPEC}/\mathsf{SELECT}\frac{\Gamma \vdash m\ \mathfrak{D}\ \texttt{sig}\ S\ \texttt{end} \qquad \mathrm{field}(m, x, S) = \mathfrak{S}}{\Gamma \vdash m.x\ \mathfrak{D}\ \mathfrak{S}}$$

Structure bodies

$$\Delta/\mathsf{S}/\mathsf{EMPTY}\frac{}{\Gamma \vdash \epsilon\ \mathfrak{DD}\ \epsilon}$$

$$\Delta/\mathsf{S}/\mathsf{DECL}\frac{\Gamma \vdash d\ \mathfrak{DD}\ \mathfrak{S} \qquad \Gamma\ ;\ x : \mathfrak{S} \vdash s\ \mathfrak{DD}\ S}{\Gamma \vdash x{:=}d; s\ \mathfrak{DD}\ x{:}\mathfrak{S}; S}$$

Fig. 3. δ-rules.

transitive closure on untyped terms is even the full relation, see proposition D.2 in appendix D. However, $\triangleright^\star$ is Church-Rosser on typed expressions. Therefore the convertibility relation is reflexive, symmetric and transitive on typed expressions.

4.4 Conversion rules

$\mathscr{MC}_2(\mathscr{S}, \mathscr{A}, \mathscr{R})$ contains two rules allowing to derive a new type for an already typed module (as well as two rules for deriving a new specification from a path already having a specification), we call them *conversion rules*.

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

The first one is linked to subtyping. It expresses that a module having a given module type M can be implicitely coerced to any supertype of M. We describe it and the associated rules for subtyping in section 4.4.1.

The second one is called the *strengthening rule*, or the *self rule*. It is similar to Leroy's strengthening rule (Leroy, 1994; Leroy, 1995). We describe it and the reasons motivating it in section 4.4.2.

4.4.1 Subtyping

In order to consider that a module of type M can be considered to be of type M' as long as M is a subtype of M', we introduce the following new rule:

$$\mathsf{M/SUB}\frac{\Gamma \vdash m : M \qquad \Gamma \vdash M' : \mathsf{modtype} \qquad \Gamma \vdash M <: M'}{\Gamma \vdash m : M'}$$

Similarly, in order to subtype specifications we introduce the following rule:

$$\mathsf{SPEC/SUB}\frac{\Gamma \vdash \mathfrak{S}' : \mathsf{wfs} \qquad \Gamma \vdash p : \mathfrak{S} \qquad \Gamma \vdash \mathfrak{S} <: \mathfrak{S}'}{\Gamma \vdash p : \mathfrak{S}'}$$

Functor types Subtyping of functor types is contravariant in the first argument:

$$\mathsf{SUB/PROD}\frac{\Gamma \vdash M_2 <: M_1 \qquad \Gamma; x_1 : M_2 \vdash M_1' <: M_2'}{\Gamma \vdash \mathtt{funcT}\ x_1 : M_1 \to M_1' <: \mathtt{funcT}\ x_2 : M_2 \to M_2'}$$

Signatures Subtyping signatures is a bit more difficult. We want a signature `sig` S_1 `end` to be a subtype of `sig` S_2 `end` if the fields given in S_2 form a subset of S_1 and for each field of S_2, the declared specification in S_2 is entailed by the specification of the corresponding field in S_1.

Therefore, in order to check $\Gamma \vdash \mathtt{sig}\ S_1\ \mathtt{end} <: \mathtt{sig}\ S_2\ \mathtt{end}$ we append S_1 to Γ, getting Γ'. Then we check that Γ' restricted to the domain of S_1 entails S_2, where "Γ' restricted to the domain of S_1 entails S_2" means that for all field name x declared with specification $\mathfrak{S}$ in S_2, we have $\Gamma' \vdash x : \mathfrak{S}$ and $x \in Dom(S_1)$, where $Dom(S)$ is defined as the set of field names declared in S.

$$\mathsf{SUB/SIG}\frac{\Gamma; S_1|_{Dom(S_1)} \vdash S_2}{\Gamma \vdash \mathtt{sig}\ S_1\ \mathtt{end} <: \mathtt{sig}\ S_2\ \mathtt{end}}$$

Entailment The rules for entailment are quite straightforward. The empty signature body is entailed by any environment and domain:

$$\mathsf{ENT/EMPTY}\frac{}{\Gamma|_{dom} \vdash \epsilon}$$

In order to check that a non-empty signature body is entailed by a given environment Γ restricted to a given domain *dom*, we first check that the first component v of the signature body has indeed specification $\mathfrak{S}$ in Γ:

$$\mathsf{ENT/DECL}\frac{x \in dom \qquad \Gamma \vdash x : \mathfrak{S} \qquad \Gamma|_{dom} \vdash S}{\Gamma|_{dom} \vdash x : \mathfrak{S}; S}$$

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

Specifications A term specification is a subspecification of an abstract specification as long as their type components are convertible:

$$\textsf{SUBSPEC/ABST}\frac{\Gamma \vdash ty(\mathfrak{S}) \bowtie t}{\Gamma \vdash \mathfrak{S} <: t}$$

A module specification is a subspecification of an abstract one as long as the type component of the former is a subtype of the type component of the latter:

$$\textsf{SUBSPEC/ABSM}\frac{\Gamma \vdash ty(\mathfrak{S}) <: M}{\Gamma \vdash \mathfrak{S} <: M}$$

Checking that a specification is a subspecification of a manifest specification is similar, but it involves checking the manifest parts are convertible:

$$\textsf{SUBSPEC/MANT}\frac{\Gamma \vdash t_1 \bowtie t'_1 \qquad \Gamma \vdash t_2 \bowtie t'_2}{\Gamma \vdash t_1 := t_2 <: t'_1 := t'_2}$$

$$\textsf{SUBSPEC/MANM}\frac{\Gamma \vdash M <: M' \qquad \Gamma \vdash m \bowtie m'}{\Gamma \vdash M := m <: M' := m'}$$

4.4.2 Strengthening

One may think the rules given so far lead to a type system for modules that can type enough module expressions. We show here a very natural example we would expect to be well-typed but that can not be typed actually if one uses only the rules given above. We then introduce a new rule, the *strengthening rule*, allowing us to type this example.

The example is the following: assume Γ is an environment containing a variable `X` declared of type ⟨*generic preorder signature*⟩. The rules we gave so far do not type `(InterPreorder X X)`, which is quite unexpected[6].

In order to see where the problem comes from, let us first consider an oversimplified example. Let

$$\Gamma = \begin{array}{l} \texttt{X : sig a:t; end;} \\ \texttt{F : funcT Y:sig a:t:=X.a; end} \rightarrow \texttt{sig end} \end{array}$$

`F` can only be applied to modules of type `sig a:t:=X.a; end`. Can we derive this type for `X`? With the typing rules given so far, the answer is no: the declared type for `X` is `sig a:t; end`, which is not a subtype of the former (the former is a subtype of the latter, not the converse). This is the problem arising with `(InterPreorder X X)`: The first application `(InterPreorder X)` is well-typed ; it is a functor expecting an argument whose component `a` is equal to `X.a`. Unfortunately, `a` is abstract in the actual argument `X` so the application of `(InterPreorder X)` to `X` fails.

However, expecting the field `a` of the type of `X` to be declared equal to `X.a` is quite natural. More generally, if the type of `X` is

$$\texttt{sig } x_1 : t_1 ; \ldots x_n : t_n ; \texttt{ end}$$

[6] At first this example can seem of little practical use: why would like one to take the intersection of a preorder with itself? However, if one want to define the equivalence relation induced by X, one has to build (*InterPreorder* (*OppPreorder X*) *X*). And the problem demonstrated in the above example arises exactly the same way. This problem also arises if one wants to build the vector space product of a vector space with itself.

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

then we would like X to have also the type

$$\texttt{sig } x_1 : t_1 := \texttt{X}.x_1 ; \ldots x_n : t_n := \texttt{X}.x_n ; \texttt{ end}$$

We say this latter signature is the former signature *strengthened by* X.

In this purpose, we add the following *strengthening rule* to $\mathcal{MC}_2(\mathcal{S}, \mathcal{A}, \mathcal{R})$. It states that if m has type M, then m also has type M/m, where M/m denotes M *strengthened by* m:

$$\mathsf{M/STR}\frac{\Gamma \vdash m : M}{\Gamma \vdash m : M/m}$$

The operator /, firstly introduced by Leroy, is called the *strengthening operator* (Leroy, 1995). We give here a slightly modified version of /:

- It transforms abstract specifications into manifest ones:

$$\begin{aligned} t/t' &= t := t' \\ M/m &= M/m := m \end{aligned}$$

- Strengthening a signature body by m means strengthening the specification of each field named x by $m.x$:

$$\begin{aligned} \epsilon/m &= \epsilon \\ (x : \mathfrak{S} ; S)/m &= x : \mathfrak{S}/m.x ; S/m \end{aligned}$$

- When strengthening a functor type by m, one has to strengthen the range of the functor by m applied to the formal argument of the functor:

$$(\texttt{funcT } x : M_1 \to M_2)/m = \texttt{funcT } x : M_1 \to (M_2/(m\ x))$$

- The other cases are defined straightforwardly:

$$\begin{aligned} (\texttt{sig } S \texttt{ end})/m &= \texttt{sig } S/m \texttt{ end} \\ (t' := t'')/t &= t' := t'' \\ (M := m')/m &= M/m := m' \end{aligned}$$

Similarly, we add a rule to strengthen specifications:

$$\mathsf{SPEC/STR}\frac{\Gamma \vdash p : \mathfrak{S}}{\Gamma \vdash p : \mathfrak{S}/p}$$

5 Metatheory

In this section, we show that $\mathcal{MC}_2(\mathcal{S}, \mathcal{A}, \mathcal{R})$ formally addresses the practical concerns we raised in section 1.1 such as safety, separate checking, independence with respect to the implementation, upward compatibility, and horizontal compatibility. More precisely, we first formalize in section 5.1 the concept of proof development, then in section 5.3 we show that $\mathcal{MC}_2(\mathcal{S}, \mathcal{A}, \mathcal{R})$ enjoys the expected properties with respect to modularity. These properties rely on strong metatheoretical results about $\mathcal{MC}_2(\mathcal{S}, \mathcal{A}, \mathcal{R})$, which we explain in section 5.3. Finally, in section 5.4, we give a correct and complete type-inference algorithm for the particular case of $\mathcal{MC}_2$ over the Calculus of Constructions, $\mathcal{MC}_2(CC)$.

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

5.1 Proof development

In practice, a proof development is a sequence of interfaces files x_1`.iv`,…, x_n`.iv` containing module types and the sequence of their respective implementation files x_1`.pv`,…, x_n`.pv` containing module expressions. We formalize this notion as follows:

Definition 5.1 (*Proof development*)
A *proof development* is a finite sequence of triples $(x_i, m_i, M_i)_{i\in[1,n]}$ where x_i is a module name, m_i is a module expression, and M_i is a module signature for all $i \in [1, n]$, and the x_i are pairwise distinct.

A given proof development is *correct* if the structure body

$$x_1 := m_1 : M_1 ; \ldots ; x_n := m_n : M_n ;$$

is well-typed, that is if we have

$$\vdash (x_1 := m_1 : M_1 ; \ldots ; x_n := m_n : M_n ;) :: (x_1 : M_1 ; \ldots ; x_n : M_n ;)$$

Modifications to a distributed proof development should always be done with care. For instance, in order not to bother the users of this development, a newer release should never drop results provided in a former one. In other words, a newer release of a proof development should always *refines* previous releases.

We formally define this refinement notion as follows:

Definition 5.2 (*Refinement of a proof development*)
Let $(x_i, m_i, M_i)_{i\in[1,n]}$ and $(x'_i, m'_i, M'_i)_{i\in[1,m]}$ be proof developments.

We say $(x_i, m_i, M_i)_{i\in[1,n]}$ is a *refinement* of $(x'_i, m'_i, M'_i)_{i\in[1,m]}$ if the three following conditions hold:

- $(x_i, m_i, M_i)_{i\in[1,n]}$ is correct,
- $(x'_i, m'_i, M'_i)_{i\in[1,m]}$ is correct,
- $\vdash$ `sig` $x_1 : M_1 ; \ldots ; x_n : M_n ;$ `end` $<:$ `sig` $x'_1 : M'_1 ; \ldots ; x'_m : M'_m ;$ `end`

In order to refine a proof development, one either changes the implementations or interfaces of some modules, or introduces some new modules.

Definition 5.3 (*Implementation changes*)
Let $(x_i, m_i, M_i)_{i\in[1,n]}$ be a correct proof development. Let j be a natural belonging to $[1, n]$ and m' be a module expression. (j, m') is an *implementation change* of $(x_i, m_i, M_i)_{i\in[1,n]}$. The *updated proof development* corresponding to this implementation change is the proof development $(x_i, m'_i, M_i)_{i\in[1,n]}$ such that

- $m'_j = m'$,
- and for all $i \in [1, n]$ with $i \neq j$, $m'_i = m_i$.

Definition 5.4 (*Interface changes*)
Let $(x_i, m_i, M_i)_{i\in[1,n]}$ be a correct proof development. Let j be a natural belonging to $[1, n]$ and M' be a module type. (j, M') is an *interface change* of $(x_i, m_i, M_i)_{i\in[1,n]}$. The *updated proof development* corresponding to this interface change is the proof

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

development $(x_i, m_i, M'_i)_{i\in[1,n]}$ such that

- $M'_j = M'$
- and for all $i \in [1,n]$ with $i \neq j$, $M'_i = M_i$.

Definition 5.5 (*Introduction of a new module*)
Let $(x_i, m_i, M_i)_{i\in[1,n]}$ be a correct proof development. Let j be a natural belonging to $[1, n+1]$, x' be a fresh variable $(\forall i \in [1,n]\ x \neq x_i)$, m' be a module expression, and M' be a module type. (j, x', m', M') is an *introduction of a new module* in $(x_i, m_i, M_i)_{i\in[1,n]}$. The *updated proof development* corresponding to this introduction is defined as the proof development $(x'_i, m'_i, M'_i)_{i\in[1,n+1]}$ such that

- $M'_j = M'$,
- for all $i \in [1, j-1]$, $M'_i = M_i$,
- and for all $i \in [j+1, n+1]$, $M'_i = M_{i-1}$.

Definition 5.6 (*One-step changes*)
Let D be a proof development. A *one-step change* of D is

- an implementation change of D,
- or an interface change of D,
- or an introduction of a new module in D.

This change is *correct* if the corresponding updated proof development is correct, it is a *refinement step* if the corresponding updated proof development is a refinement of D.

One may wonder whether these three operations are enough in practice. It is too soon to answer this question, however the current practice in programming languages gives us some hints here. The above operations indeed correspond to the development model of Objective Caml. It seems to be quite satisfactory for medium-sized development. However, for very large developments, and especially when providing large libraries, users seem to need more: an operation users request more and more on the Objective Caml mailing list is the ability to pack several modules into one.

For the moment, we choose to restrict our study to the three operations above. Adding the packing operation does not seem to raise any metatheoretical problem and should be quite feasible in practice.

5.2 Formal guarantees with respect to modular development

In the previous section, we defined what a proof development is and we defined a notion of a refinement step in the life cycle of a proof development.

We can now wonder how these notions behave with respect to modularity concerns:

- Can we check that a development is correct in a modular way?
- Can we check that a given one-step change is a refinement locally, that is without checking that the newer development obtained by this step is a refinement of its former version?

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

- Can we guarantee that modules do not introduce any inconsistency in the proof system they extend?

In this section, we answer these questions positively. This shows that the practical concerns we mentioned section 1.1 are addressed by $\mathscr{MC}_2(\mathscr{S},\mathscr{A},\mathscr{R})$. The proof we give here are based on the metatheoretical results given in the more technical section 5.3.

The following is a criterion for checking the correctness of a proof development in a modular way.

Proposition 5.7 (*Separate checking*)
A proof development $(x_i, m_i, M_i)_{i\in[1,n]}$ is correct if and only if, for all $i \in [1,n]$, we have

$$x_1 : M_1; \ldots ; x_{i-1} : M_{i-1} \vdash m_i : M_i$$

Proof
This is a direct consequence of the rules for typing structure bodies. The proof is done by induction on n. □

Therefore, the correctness of a proof can be checked modularly. This allows proof development to be conducted independently by several developers once they have agreed on interfaces.

We can now formalize the fact that $\mathscr{MC}_2(\mathscr{S},\mathscr{A},\mathscr{R})$ enjoys independence with respect to the implementation, upward compatibility and horizontal compatibility properties introduced in section 1.1. These properties are precisely local criteria for checking that a given one-step change is a refinement.

Proposition 5.8 (*Independence with respect to the implementation*)
Let $(x_i, m_i, M_i)_{i\in[1,n]}$ be a correct proof development. Let (j, m') be an implementation change of $(x_i, m_i, M_i)_{i\in[1,n]}$. (j, m') is correct and is in fact a refinement step if and only if we have

$$x_1 : M_1; \ldots ; x_{j-1} : M_{j-1} \vdash m' : M_j$$

Proof
This is a direct consequence of proposition 5.7. □

Proposition 5.9 (*Upward compatibility*)
Let $(x_i, m_i, M_i)_{i\in[1,n]}$ be a proof development. Let (j, M') be an interface change of it. (j, M') is a refinement step if we have

$$x_1 : M_1; \ldots ; x_{j-1} : M_{j-1} \vdash m_j : M'$$

and

$$x_1 : M_1; \ldots ; x_{j-1} : M_{j-1} \vdash M' <: M_j$$

Proof
Assume $(x_i, m_i, M_i)_{i\in[1,n]}$ is a correct proof development. Let (j, M') be a correct interface change of it, and $(x_i, m_i, M'_i)_{i\in[1,n]}$ be the updated proof development. By proposition 5.7, for all $i \in [1,n]$, we have

$$x_1 : M_1; \ldots ; x_{i-1} : M_{i-1} \vdash m_i : M_i$$

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

Therefore, by definition of M'_i, for all $i < j$, we have

$$x_1 : M'_1; \ldots ; x_{i-1} : M'_{i-1} \vdash m_i : M'_i$$

Moreover, as (j, M') is a locally correct interface change, we have

$$x_1 : M'_1; \ldots ; x_{j-1} : M'_{j-1} \vdash m_j : M'_j$$

At last, since $x_1 : M'_1; \ldots ; x_{j-1} : M'_{j-1} \vdash M'_j <: M_j$, applying the weakening lemma 5.14, for all $j > i$ we get

$$x_1 : M'_1; \ldots ; x_{i-1} : M'_{i-1} \vdash m_i : M'_i$$

Therefore, by proposition 5.7, $(x_i, m_i, M'_i)_{i\in[1,n]}$ is a correct proof development. Moreover, by hypothesis and lemma 5.13, we have

$$x_1 : M'_1; \ldots ; x_n : M'_n \vdash M'_j <: M_j$$

so that, by rule [M/SUB], we have

$$x_1 : M'_1; \ldots ; x_n : M'_n \vdash x_j : M_j$$

For all $i \neq j$, we also have trivially

$$x_1 : M'_1; \ldots ; x_n : M'_n \vdash x_j : M_j$$

Therefore, for all $i \in [1, n]$, we have

$$x_1 : M'_1; \ldots ; x_n : M'_n|_{\{x_1,\ldots,x_n\}} \vdash x_1 : M_1 ; \ldots ; x_n : M_n ;$$

Therefore, (j, M') is a refinement step. □

Note that the upward compatibility property generally does not hold in programming languages such as Standard ML or Objective Caml, because of the `open` construct, as shown in Section 3.5.2.

Proposition 5.10 (*Horizontal compatibility property*)
Let $(x_i, m_i, M_i)_{i\in[1,n]}$ be a correct proof development. Let (j, x', m', M') be an introduction of a new module in it. (j, x', m', M') is a refinement step if and only if we have

$$x_1 : M_1; \ldots ; x_{j-1} : M_{j-1} \vdash m' : M'$$

Proof
This follows trivially from proposition 5.7 and lemma 5.13. □

Several Pure Type Systems are known as rigorous formal basis for developing proofs as they have been proved logically consistent. This is the case for instance of the Calculus of Construction (Coquand & Gallier, 1990). One may wonder whether the module layer we add to a given PTS might endanger this consistency or even if it changes its logical expressive power. Fortunately, the answer is no for $\mathcal{MC}_2(CC)$:

Proposition 5.11 (*Conservativity of* $\mathcal{MC}_2(CC)$)
$\mathcal{MC}_2(CC)$ is conservative. More precisely assume Γ is an environment containing no module declaration, assume no module expressions appears in the specifications

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

contained in Γ, and assume t is a base term expression containing no module expressions. If t is an inhabited type in $\mathscr{MC}_2(CC)$, *i.e.* if there exists t' such that $\Gamma \vdash t' : t$ is derivable in $\mathscr{MC}_2(CC)$, then there exists a base term expression t'', containing no module expression, such that $\Gamma \vdash t'' : t$ is derivable in CC.

Proof
This theorem is a corollary of the subject-reduction and strong normalization theorems (theorems 5.15 and 5.17, below): as t' normalizes by $\triangleright_{\beta_m\rho\delta}$, let t'' be a normal form of it ; $\Gamma \vdash t'' : t$ is derivable in $\mathscr{MC}_2(CC)$. Consider the derivation of this judgment we get by applying the strategy given by the algorithmic rules given section 5.4. As no module expression appears in t'' nor in Γ nor in t then this derivation uses only PTS rules and the rule [SPEC/VAR]. Therefore, it can be derived in CC. □

Remark 5.12
Although we did not prove it formally, we conjecture this result still holds for other PTS. The difficulty here lies more in the PTS structure than in $\mathscr{MC}_2$ itself: the problem is the rule [T/LAM], like for the well-known expansion postponement problem (van Benthem Jutting *et al.*, 1993).

5.3 Metatheoretical foundations

The modular properties of $\mathscr{MC}_2(\mathscr{S},\mathscr{A},\mathscr{R})$ expressed in the previous section are no accident. Indeed they are rooted in very strong metatheoretical properties of this calculus. We give in this section the salient metatheoretical results of $\mathscr{MC}_2(\mathscr{S},\mathscr{A},\mathscr{R})$. In section 5.3.1, we give the usual weakening results. In section 5.3.2, we present the main results about reduction in $\mathscr{MC}_2(\mathscr{S},\mathscr{A},\mathscr{R})$ and we briefly sketch the proof method we used to prove them.

5.3.1 Weakening

$\mathscr{MC}_2(\mathscr{S},\mathscr{A},\mathscr{R})$ enjoys the usual weakening lemma of type systems:

Lemma 5.13 (*Weakening*)
Let Γ and Γ' be any two environments. Let γ be any expression and γ' be wfs or sigbody or modtype or any expression. Let M be any module type and x be a variable not appearing in Γ nor Γ'. If

$$\Gamma;\Gamma' \vdash \gamma : \gamma'$$

then

$$\Gamma; x : M; \Gamma' \vdash \gamma : \gamma'$$

Moreover, a judgment may also be weakened by subtyping:

Lemma 5.14 (*Weakening by subtyping*)
Let Γ and Γ' be any two environments. Let γ be any expression and γ' be wfs or sigbody or modtype or any expression. Let M and M' be any two module types

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

and x be a variable not appearing in Γ nor Γ'. If

$$\begin{cases} \Gamma \vdash M : \mathsf{modtype} \text{ and} \\ \Gamma \vdash M' <: M \text{ and} \\ \Gamma; x : M; \Gamma' \vdash \gamma : \gamma' \end{cases}$$

then

$$\Gamma; x : M'; \Gamma' \vdash \gamma : \gamma'$$

5.3.2 Properties of the reduction

The three expected main metatheoretical properties for $\mathcal{MC}_2(\mathcal{S}, \mathcal{A}, \mathcal{R})$ hold. We give here only a brief sketch of their proof; the interested reader may consult our research report for the details (Courant, 1999).

The subject-reduction statement is a straightforward generalization of subject-reduction theorems for typed lambda-calculi to env-dependent reduction relations.

Theorem 5.15 (*Subject-reduction*)
Let Γ be any environment. Let γ and γ' be any two expressions and γ'' be wfs or sigbody or modtype or any expression. Assume $\Gamma \vdash \gamma \triangleright^\star \gamma'$ and $\Gamma \vdash \gamma : \gamma''$. Then we have $\Gamma \vdash \gamma : \gamma'$.

Similarly, the statement of the Church-Rosser property is a straightforward generalization of the Church-Rosser property to an env-dependent reduction relation.

Theorem 5.16 (*Church-Rosser property*)
$\triangleright^\star$ is Church-Rosser on typed terms. More formally, let Γ be an environment such that $\Gamma \vdash \mathsf{ok}$ and $\gamma, \gamma_1, \gamma_2$ be any three expressions such that γ is well-typed, $\Gamma \vdash \gamma \triangleright^\star \gamma_1$ and $\Gamma \vdash \gamma \triangleright^\star \gamma_2$. Then there exists γ' such that $\Gamma \vdash \gamma_1 \triangleright^\star \gamma'$ and $\Gamma \vdash \gamma_2 \triangleright^\star \gamma'$.

Theorem 5.17 (*Strong normalization*)
$\triangleright_{\beta_m \rho \delta}$ is strongly normalizing on well-typed terms. Formally: let Γ be an environment such that $\Gamma \vdash \mathsf{ok}$ and γ_1 is well-typed, then any sequence $\gamma_1, \ldots, \gamma_n, \ldots$ verifying $\forall i \; \Gamma \vdash \gamma_i \triangleright \gamma_{i+1}$ is a finite sequence.

Remark 5.18
The normalization result is only for $\triangleright_{\beta_m \rho \delta}$, not for $\triangleright$, as $\triangleright$ can be non-normalizing: if β-reduction in the base PTS is non-normalizing, $\triangleright$ is of course non-normalizing. If β-reduction is normalizing, from the normalization of $\triangleright_{\beta_m \rho \delta}$ and $\triangleright_\beta$, one can trivially derive a weak normalization result for $\triangleright$ (as it is their union). We conjecture that the strong normalization of $\triangleright$ can be proved if the base PTS is the Calculus of Constructions; however this would much complicate the normalization proof. Whether the strong normalization of the base PTS implies the strong normalization of $\triangleright$ in general is an open question.

The proof of the Church-Rosser property is quite complex: it relies on the strong normalization of typed expressions and even appears to be as complex as the normalization proof itself. Therefore, we chose to prove at once the Church-Rosser

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

property (for $\beta_m\rho\delta$-reduction, modulo β-equivalence) and strong normalization of the $\beta_m\rho\delta$ reduction over typed expressions.

This raises another problem: with the presentation given in this paper, the Church-Rosser property is required in order to prove for instance the transitivity of subtyping (since this latter relies on the transitivity of convertibility). Unfortunately, the subject-reduction property relies on this transitivity property. A possible solution to treat this could be to prove the Church-Rosser property, strong normalization and subject-reduction at once, in the style of Goguen's normalization proof for UTT (Goguen, 1994). We recently showed such an approach was possible in the simpler framework of singleton types (Courant, 2002b). However we chose another approach (Courant, 1999): we explicitly added a transitivity rule for convertibility. In order to prevent the problem we have with the untyped conversion, this rule is put under type conditions. Then we could prove the subject-reduction first, and then the strong normalization and Church-Rosser property at once.

5.4 Type inference

In this section, we present the rules for type inference in $\mathscr{MC}_2(CC)$. As for base terms, type inference for PTS is a quite complex subject (van Benthem Jutting *et al.*, 1993) and goes beyond the scope of this paper. For the sake of simplicity we therefore restrict ourselves to the Calculus of Constructions (which is a full functional PTS) for our base terms and focus on the typing rules for module expressions. However, we believe the usual systems for typechecking different classes of PTS (van Benthem Jutting *et al.*, 1993) could be lifted to $\mathscr{MC}_2(\mathscr{S},\mathscr{A},\mathscr{R})$.

As usual for type systems involving subtyping, the inference rules defining $\mathscr{MC}_2(CC)$ are non-deterministic. However, as every well-typed module expression has a principal type (most general type), type-checking is quite simple and mainly amounts to enforce the right strategy over the application of the rules.

Typing a module expression is done by computing its principal type (also called most general type); checking that the module expression belongs to some given type is done by checking that its principal type belongs to the latter.

We introduce some new judgments, whose rules are deterministic, in order to type-check module expressions:

- $\Gamma \vdash m :_{\mathscr{P}} M$, read "in the environment Γ, m has principal type M"
- $\Gamma \vdash p :_{\mathscr{P}} \mathfrak{S}$, read "in the environment Γ, p has principal specification $\mathfrak{S}$"
- $\Gamma \vdash m :_c M$ ("algorithmically m has type M") which is defined as the algorithmic counterpart of the previously used judgment $\Gamma \vdash m : M$. ("m has type M"). The intent is that m has type M if and only if algorithmically m has type M.

Similarly, we introduce

- $\Gamma \vdash p :_c \mathfrak{S}$,
- $\Gamma \vdash M <:_c M'$,
- $\Gamma \vdash \mathfrak{S} <:_c \mathfrak{S}'$,
- $\Gamma \vdash M :_c \mathsf{modtype}$,

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

- $\Gamma \vdash S :_c \mathsf{sigbody}$,
- $\Gamma \vdash \mathfrak{S} :_c \mathsf{wfs}$ and
- $\Gamma|_{dom} \vdash_c S$

as the respective algorithmic counterparts of $\Gamma \vdash p : \mathfrak{S}$, $\Gamma \vdash M <: M'$, $\Gamma \vdash \mathfrak{S} <: \mathfrak{S}'$, $\Gamma \vdash M : \mathsf{modtype}$, $\Gamma \vdash S : \mathsf{sigbody}$, $\Gamma \vdash \mathfrak{S} : \mathsf{wfs}$ and $\Gamma|_{dom} \vdash S$.

Well-formedness Checking module types, module signatures and subtyping is done as in the non-algorithmic system. See figure A 1 and A 2 in appendix A.

Checking a module belongs to a given type Checking that a module expression belongs to some given type is just a matter of applying the rule [M/SUB], which is the only rule for the judgment $\Gamma \vdash m :_c M$:

$$\mathsf{M/SUB}\frac{\Gamma \vdash m :_{\mathscr{P}} M \qquad \Gamma \vdash M' :_c \mathsf{modtype} \qquad \Gamma \vdash M <:_c M'}{\Gamma \vdash m :_c M'}$$

Checking a path belongs to some specification is just a matter of applying the rule [SPEC/SUB], which is the only rule for the judgment $\Gamma \vdash p :_c \mathfrak{S}$:

$$\mathsf{SPEC/SUB}\frac{\Gamma \vdash \mathfrak{S}' :_c \mathsf{wfs} \qquad \Gamma \vdash p :_{\mathscr{P}} \mathfrak{S} \qquad \Gamma \vdash \mathfrak{S} <:_c \mathfrak{S}'}{\Gamma \vdash p :_c \mathfrak{S}'}$$

Inferring the principal type of a module As we have seen, strengthening might be needed in some cases to compute the most general type of an expression. Therefore, in order to compute the principal type of a module expression m, we first apply the rule whose conclusion syntactically matches m, getting a type M we call the *quasi-principal* type of m. Then, we apply strengthening to this type. Similarly, we introduce a notion of *quasi-principal* specification for module paths. In other words, we introduce the following new judgments:

- $\Gamma \vdash m :_{qp} M$, read "in Γ, m has quasi-principal type M",
- $\Gamma \vdash p :_{qp} \mathfrak{S}$, read "in Γ, p has quasi-principal specification $\mathfrak{S}$",
- $\Gamma \vdash s ::_{qp} S$, read "in Γ, the structure body s has quasi-principal type S",
- and $\Gamma \vdash d ::_{qp} \mathfrak{S}$, read "in Γ, the definition d has quasi-principal specification $\mathfrak{S}$".

The inference rules for these judgments are given figure 4.

Actually, the only rules for principal typing are the strengthening rules for module expressions and module paths:

$$\mathsf{M/STR}\frac{\Gamma \vdash m :_{qp} M}{\Gamma \vdash m :_{\mathscr{P}} M/m}$$

$$\mathsf{SPEC/STR}\frac{\Gamma \vdash p :_{qp} \mathfrak{S}}{\Gamma \vdash p :_{\mathscr{P}} \mathfrak{S}/p}$$

Base terms Finally, the algorithmic rules for the Calculus of Constructions are given figure 5. We make use of three new judgments:

- $\Gamma \vdash t :_c t'$, used to check that t' is a type of t,

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

Module expressions

$$\mathsf{M/STRUCT}\frac{\Gamma \vdash s ::_{qp} S \qquad \mathrm{Alldiff}(S)}{\Gamma \vdash \texttt{struct}\ s\ \texttt{end} :_{qp} \texttt{sig}\ S\ \texttt{end}}$$

$$\mathsf{M/APP}\frac{\Gamma \vdash m_1 :_{qp} \texttt{funcT}\ x{:}M \to M' \qquad \Gamma \vdash m_2 :_c M}{\Gamma \vdash (m_1\ m_2) :_{qp} M'\{x \leftarrow m_2\}}$$

$$\mathsf{M/LAM}\frac{\Gamma \vdash M :_c \texttt{modtype} \qquad \Gamma; x : M \vdash m :_{qp} M'}{\Gamma \vdash \texttt{functor}\ x{:}M \to m :_{qp} \texttt{funcT}\ x{:}M \to M'}$$

$$\mathsf{M/PATH}\frac{\Gamma \vdash p :_{qp} \mathfrak{S}}{\Gamma \vdash p :_{qp} ty(\mathfrak{S})}$$

Structure bodies

$$\mathsf{S/EMPTY}\frac{}{\Gamma \vdash \epsilon ::_{qp} \epsilon} \qquad \mathsf{S/DECL}\frac{\Gamma \vdash d ::_{qp} \mathfrak{S} \qquad \Gamma\ ;\ x : \mathfrak{S} \vdash s ::_{qp} S}{\Gamma \vdash x{:=}d; s ::_{qp} x{:}\mathfrak{S}; S}$$

Definitions

$$\mathsf{DEF/M1}\frac{\Gamma \vdash m :_{\mathscr{P}} M}{\Gamma \vdash m ::_{qp} M} \qquad \mathsf{DEF/M2}\frac{\Gamma \vdash m :_c M}{\Gamma \vdash m{:}M ::_{qp} M}$$

$$\mathsf{DEF/TRANSPM}\frac{\Gamma \vdash m :_{\mathscr{P}} M}{\Gamma \vdash \texttt{transparent}\ m ::_{qp} M{:=}m}$$

$$\mathsf{DEF/TERM}\frac{\Gamma \vdash t_1 :_{\mathscr{P}} t_2}{\Gamma \vdash t_1 ::_{qp} t_2{:=}t_1}$$

Paths

$$\mathsf{SPEC/VAR}\frac{}{\Gamma \vdash x :_{qp} \Gamma(x)}$$

$$\mathsf{SPEC/SELECT}\frac{\Gamma \vdash m :_{qp} \texttt{sig}\ S\ \texttt{end} \qquad \mathrm{field}(m, x, S) = \mathfrak{S}}{\Gamma \vdash m.x :_{qp} \mathfrak{S}}$$

Fig. 4. Rules for quasi-principal typing in $\mathscr{MC}_2(CC)$.

- $\Gamma \vdash t :_{\mathscr{P}} t'$, used to infer one type of t
- and $\Gamma \vdash t :_w t'$, used to infer a type in weak-head normal form for t.

5.4.1 *Properties of the rules for type inference*

Notice that if you replace in each rule the algorithmic judgment by its non-algorithmic counterpart, you get exactly the non-algorithmic rules of $\mathscr{MC}_2(CC)$, as presented in section 4.2.3, except for the rule [T/WHNF] whose non-algorithmic counterpart is [T/CONV].

Proposition 5.19 (*Soundness*)
Let Γ be such that $\Gamma \vdash \mathsf{ok}$. For all m and M, if

- $\Gamma \vdash m :_{\mathscr{P}} M$ or

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

$$\mathsf{T/PATH}\frac{\Gamma \vdash p :_{qp} \mathfrak{S}}{\Gamma \vdash p :_{\mathscr{P}} ty(\mathfrak{S})}$$

$$\mathsf{T/SORT}\frac{(\sigma_1, \sigma_2) \in \mathscr{A}}{\Gamma \vdash \sigma_1 :_{\mathscr{P}} \sigma_2}$$

$$\mathsf{T/LAM}\frac{\Gamma \vdash t_1 :_w \sigma_1 \qquad \Gamma; x : t_1 \vdash t_2 :_{\mathscr{P}} t_3 \qquad \Gamma \vdash \forall x{:}t_1 . t_3 :_{\mathscr{P}} \sigma_2}{\Gamma \vdash \lambda x{:}t_1 . t_2 :_{\mathscr{P}} \forall x{:}t_1 . t_3}$$

$$\mathsf{T/PROD}\frac{\Gamma \vdash t_1 :_w \sigma_1 \qquad \Gamma; x : t_1 \vdash t_2 :_w \sigma_2 \qquad (\sigma_1, \sigma_2, \sigma_3) \in \mathscr{R}}{\Gamma \vdash \forall x{:}t_1 . t_2 :_{\mathscr{P}} \sigma_3}$$

$$\mathsf{T/APP}\frac{\Gamma \vdash t_1 :_w \forall x{:}t_4 . t_5 \qquad \Gamma \vdash t_2 :_c t_4}{\Gamma \vdash (t_1\ t_2) :_{\mathscr{P}} t_5\{x \leftarrow t_2\}}$$

$$\mathsf{T/WHNF}\frac{\Gamma \vdash t_1 :_{\mathscr{P}} t_2 \qquad \Gamma \vdash t_2 \triangleright^{\mathrm{whnf}} t_3}{\Gamma \vdash t_1 :_w t_3}$$

$$\mathsf{T/CONV}\frac{\Gamma \vdash t_1 :_{\mathscr{P}} t_3 \qquad \Gamma \vdash t_2 :_w \sigma \qquad \Gamma \vdash t_2 \bowtie t_3}{\Gamma \vdash t_1 :_c t_2}$$

Fig. 5. Type checking base terms in $\mathscr{MC}_2(CC)$.

- $\Gamma \vdash m :_{qp} M$ or
- $\Gamma \vdash m :_c M$

then $\Gamma \vdash m : M$.

Proposition 5.20 (*Completeness*)
Let Γ be such that $\Gamma \vdash \mathsf{ok}$. For all m and M, if $\Gamma \vdash m : M$ then

- $\Gamma \vdash m :_c M$
- there exists a unique M' such that $\Gamma \vdash m :_{\mathscr{P}} M'$. Moreover, $\Gamma \vdash M' <:_c M$.
- there exists a unique M' such that $\Gamma \vdash m :_{qp} M'$. Moreover, $\Gamma \vdash M'/m <:_c M$.

Corollary 5.21 (*Principal type*)
Every well-type module expression has a most general module type, also called principal type.

Finally, the reader can check that the rules we have given are deterministic. The conversion test involved in rules [SUBSPEC/MANM], [SUBSPEC/MANT] and [SUBSPEC/ABST] is decidable as $\triangleright$ is Church-Rosser and normalizing (see theorems 5.16 and 5.17). Therefore, we have an algorithm to decide the type of a given module expression.

6 Implementations

$\mathscr{MC}_2$ can be seen as an *a posteriori* formal justification of a subset of the proof-assistant Agda (Coquand, 2000). It has also been used as the formal basis of the module system of Coq, implemented by Jacek Chrząszcz (Chrząszcz, 2004b; Chrząszcz, 2004a; Chrząszcz, 2003). Roughly, this implementation accepts $\mathscr{MC}_2$

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

expressions, where module paths are restricted to identifiers and accesses of fields of module paths.

As a proof of concept, we also developed a prototype type-checker of $\mathscr{MC}_2(CC)$, called **oeuf**. **oeuf** accepts exactly $\mathscr{MC}_2(CC)$ expressions. **oeuf** notably features separate verification of modules, as presented in Section 5.

6.1 Faithfulness to the theory

When we wrote **oeuf**, our main concern was to ensure consistency between our implementation and the rules we describe here. In order to achieve this, we wrote a rule compiler, **tpresent**, able to compile first-order inference rules into Objective Caml code as well as into LaTeX code. All rules presented in this paper have been generated that way.

The translation process into LaTeX being quite flexible, we could hide some implementation details. This does not alter the consistency between the printed and the implemented rules as we can precisely know what the differences are by a simple look at the rules governing the translation. The only differences are:

- As the mathematical presentation of a rule can harmlessly be a little more ambiguous than its implementation, we chose to use the same notation for typing judgments for terms, for modules and for paths. Likewise, while the implementation distinguishes term specifications and module specifications, providing specification as their disjoint sum, we chose not to write the canonical injection from term specifications to specifications nor the one from module specifications to specifications.
- For two of the given rules, we chose to hide some information that would have made them less readable. The first one is the rule [SUB/SIG] ; as we use De Bruijn indices, we needed to lift an expression once in this rule:

$$\textsf{SUB/SIG}\frac{\Gamma; S_1|_{Dom(S_1)} \vdash_c \uparrow^{|S_1|} S_2}{\Gamma \vdash \texttt{sig}\ S_1\ \texttt{end} <:_c \texttt{sig}\ S_2\ \texttt{end}}$$

  where $\uparrow^n S$ denotes the signature S lifted n times. The second one is the rule [ENT/DECL]. Because of the renaming problem discussed in appendix C, *dom* in the judgment $\Gamma|_{dom} \vdash_c S$ is not a set of fields but rather a table from field names to De Bruijn indices. Likewise $Dom(S)$ denotes the table obtained for S. Then, the rule ENT/DECL is in reality

$$\textsf{ENT/DECL}\frac{(w,x) \in dom \qquad \Gamma \vdash x :_c \mathfrak{S} \qquad \Gamma|_{dom} \vdash_c S\{w \leftarrow x\}}{\Gamma|_{dom} \vdash_c w : \mathfrak{S}; S}$$

6.2 Bells and Whistles

The concrete syntax of the input files is close to that given in section 4.2 with very few differences:

- The notations "λ", "$\forall$", "$\rightarrow$" have been replaced by the ASCII strings "\", "!" and ->.

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

- Products and abstractions can be done on several variables at once, thus one can write `\x y : nat . y` instead of `\x : nat . \y : nat . y`.
- Variables whose names contain non-alphanumeric characters are infix (thus, `<=` is infix);
- Any term can be considered infix as long as it appears between backquotes.

Also, we added local definitions (`let ...in` expressions) to base terms, using the rules found in (Severi, 1996). The rules have not been given in this paper for the sake of simplicity.

6.3 Separate development

`oeuf` knows three kinds of files:

Interfaces these are files with a name ending with `.iv`; they should contain a module type;

Implementations these are files with a name ending with `.pv`; they should contain a module expression;

Compiled interfaces these are files with a name ending with `.civ`; they should contain a compiled representation of an interface.

Compiling a f`.iv` file with **oeuf** makes **oeuf** check the module type contain in f`.iv` in the environment built with the interfaces contained in the compiled interfaces of the current directory.

Compiling a f`.pv` file with **oeuf** makes **oeuf** typecheck the module expression contained in f`.pv` in the same environment. Moreover, **oeuf** checks that a compiled interface f`.civ` exists and that the inferred type for the contents of f`.pv` is a subtype of it.

Then, if one wants to check a proof development $(x_i, m_i, M_i)_{i\in[1,n]}$ (in the sense of definition 5.1), one just has to write each m_i in a file named x_i`.pv`, each M_i in a file x_i`.iv`, compile successively x_1`.iv`,...,x_n`.iv`, then compile all the x_i`.pv` in any order.

6.4 Availability

oeuf is available on the World-Wide-Web, at `http://www-verimag.imag.fr/~courant/soft/oeuf/`

7 Future work

In this section, we describe several extensions we would like to add to $\mathscr{MC}_2$ in decreasing order of need.

7.1 Inductive types

Adding inductive types such as those found in Coq to our calculus does not seem too difficult. However, this would raise interesting issues about the status of inductive types.

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

In Coq, until version V5.8, inductive types were first-class objects and isomorphic inductive types were identified. This semantics would fit well within $\mathscr{MC}_2$. However, this semantics has an annoying drawback: if the user defines an inductive with a name, and the reduction machine unfolds this name, the user then face a first-class inductive type, which is quite confusing.

In order to solve this problem, the semantics changed since V5.10: inductive type definitions became generative, that is an inductive type definition always generates a new type, incompatible with all previously defined types. Unfortunately, this semantics does not fit well within $\mathscr{MC}_2$ since a type definition in a structure can potentially be duplicated by reduction; in other words, this semantics would make $\mathscr{MC}_2$ lack the subject-reduction property.

Let us explain this with an example. Consider first the following functor definition:

```
module F :=
  functor X : sig
               t : Set;
               f : t → t;
               v : t;
              end
   →
    struct
     Y := X; (* Y.t = X.t *)
     Z := X; (* Z.t = X.t *)
     r := Y.f Z.v; (* ok since Y.t = X.t = Z.t *)
    end
```

Now, assuming such inductive type definitions exist in $\mathscr{MC}_2$, let us apply this module to an anonymous structure defining `t` as an inductive definition:

```
module APP := F(struct
                inductive t : Set := A : Set | B : Set;
                f :=  x : A . x;
                v := A;
               end)
```

Now, if we replace the formal parameter of `F` by its actual argument, we get the following module:

```
APP' :=
 struct
  Y :=
   struct
    inductive t : Set := A : t | B : t;
    f := λ x : t . x;
    v := A;
   end
  module Z =
   struct
```

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

```
    inductive t : Set := A : t | B : t;
    f := λ x : t . x;
    v := A;
   end
   r := Y.f Z.v;
 end
```

In the former V5.8 version of Coq, inductive types are first-class object (there are anonymous inductive types), and an inductive definition is just a definition binding a name to an anonymous inductive type. Any isomorphic types are considered equal. Therefore, since `APP'.Y.t` and `APP'.Z.t` have the same definition, they are equal types. On the contrary, starting with the V5.10 release, they are inductive types with different names, hence they are considered different — as in ML — which implies the definition of `APP'.r` is ill-typed.

However the V5.8 has two problems:

- First, it identifies too many types. For instance, with the following inductive definitions, `nat` and `X` are considered equal:

  ```
  Inductive nat : Set := O : nat | S : nat -> nat.
  Inductive X : Set := A : X | B : X -> X.
  ```

  Moreover, the order in which the constructors are given is significant (if the declaration of `A` and `B` were swapped in the definition of `X`, `X` and `nat` would be considered different) which is not nice from a software engineering point of view: equality should be intentionnal, not accidental.
- Second, as you prefer to show inductive names rather than the anonymous inductive they are bound to, Coq V5.8 has to deal carefully with reductions.

The variants types of OCaml are better with respect to the former issue (they identify types by name), and they are compatible with subject-reduction:

```
module APP' =
 struct
  module Y =
   struct
    type t = [‘A | ‘B]
    let f (x:t) = x
    let v = ‘A
   end
  module Z =
   struct
    type t = [‘A | ‘B]
    let f (x:t) = x
    let v = ‘A
   end
  let x = Y.f Z.v
 end
```

is accepted by OCaml. However they do not solve the latter issue.

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

We conjecture there is a solution to the subject-reduction problem where inductive types are not first-class objects, hence solving both problems:

Although in order to make `APP'` well-typed, `Y.t` and `Z.t` should convert in

```
module Y = struct inductive t : Set := A : t | B : t end;
module Z = struct inductive t : Set := A : t | B : t end;
```

there seems to exists an equality notion that is compatible with subject-reduction and that does not make `Y.t` and `Z.t` convert in

```
module Y := struct inductive t : Set := A : t | B : t end
         : sig inductive t : Set := A : t | B : t end;
module Z : struct inductive t : Set := A : t | B : t end
         : sig inductive t : Set := A : t | B : t end;
```

(Whereas Objective Caml would consider them as equal types if `t` is declared as a variant type.)

The idea is in the former case, `Y` and `Z`, would both have a signature telling `t` is equal to `(struct inductive t : Set := A : t | B : t end).t`, the normal form `Y.t` and `Z.t` would reduce to, whereas in the latter case, they would just have a signature telling `t` is an inductive type build with `A` and `B`, and `Y.t` and `Z.t` would be different, hence incompatible, normal form.

7.2 Hierarchy of universes

Adding a cumulative hierarchy of universes in the style of (Luo, 1990) to our calculus seems easy since $\mathscr{MC}_2$ is quite independent from the base language. However, we would like more: we would like to have some universe polymorphism. The current way Coq deals with universes polymorphism is not satisfying from the point of view of modularity (Asperti, 1999). This is due partly to Coq not fully implementing typical ambiguity (Harper & Pollack, 1991), but also to separate checking requiring explicit universe constraints. We recently proposed an extension of ECC with explicit polymorphic universes (Courant, 2002a) and we implemented it in our prototype **oeuf**. Further work is needed to see whether these explicit universes are convenient for daily use.

7.3 Overloaded Functors

María Virginia Aponte and Giuseppe Castagna proposed the addition of overloaded functors to SML (Aponte & Castagna, 1996). Such functors are providing implementations of structures that depend on the actual parameters they are fed with. For instance, a functor implementing dictionaries may implement them as associations lists when it is only applied to a structure defining only a type and an equivalence relation over this type, and as balanced binary trees when applied to a structure defining a type together with a comparison function over this type.

These functors should help managing the namespace when dealing with large developments. Indeed remembering names would be easier since names could be

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

overloaded. Moreover, it would help reusing code since the resolution of overloading is done at run-time.

These overloaded functors seem to be especially interesting when developing abstract algebra. Many operations on algebraic structures indeed share the same name, although they are distinct. For instance the name "product" applies to monoids, groups, rings, vector spaces; the "polynomial ring of" functor produces a ring when applied to a ring, an integral domain when applied to an integral domain.

Unfortunately, no subject-reduction result could be given for the proposal of Aponte and Castagna as the underlying module system does not enjoy the subject-reduction property. We hope our work provides a framework in which this proof can be done.

8 Conclusion

We identified some issues related to the separate development of large proofs such as independence with respect to the implementation, horizontal compatibility, upward compatibility and composability. We introduced $\mathscr{MC}_2$, a kernel module system for proof languages based on Pure Type Systems aimed at giving a formal basis for solving these issues. $\mathscr{MC}_2$ over the Calculus of Construction is the formal basis upon which modules have been implemented in Coq (Chrząszcz, 2004b; Chrząszcz, 2004a; Chrząszcz, 2003). It can also be seen as a justification of a subset of the proof-assistant Agda.

Through basic examples, we showed how $\mathscr{MC}_2$ helps structuring the development of mathematical structures and theories.

We formalized the issues related to separate development as metatheoretical properties of $\mathscr{MC}_2$ and proved it enjoys them. $\mathscr{MC}_2$ also enjoys metatheoretical properties making it a safe basis for proof development: it has a reduction semantics (subject-reduction holds), it enjoys the Church-Rosser property, and it strongly normalizes. $\mathscr{MC}_2(CC)$ — $\mathscr{MC}_2$ over the Calculus of Constructions — is a conservative extension of CC. We have shown that type inference is decidable in $\mathscr{MC}_2(CC)$.

Moreover, as $\mathscr{MC}_2$ is quite independent from base terms, it is likely to be robust with respect to changes in the base calculus (such as the addition of inductive types or universes).

Finally, the idea of defining a module system for proof languages incorporating the "manifest field" feature of SML-like module systems is quite original as far as we know. Therefore, it opens many directions for future work.

A $\mathscr{MC}_2(CC)$ type inference algorithm

We give figure A 1 the algorithmic inference rules for checking the formation of module types and specifications, and figure A 2 the rules for checking subtyping.

We recall figure A 3 the rules for quasi-principal typing already given figure 4 and give figure A 4 the rules for inferring the principal type of a module, for inferring the principal specification of a path, for checking a module has a given type, and for checking a path has a given specification.

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

Module types

$$\mathsf{SB/EMPTY}\frac{}{\Gamma \vdash \epsilon :_c \mathsf{sigbody}}$$

$$\mathsf{SB/DECL}\frac{\Gamma \vdash \mathfrak{S} :_c \mathsf{wfs} \qquad \Gamma; x : \mathfrak{S} \vdash S :_c \mathsf{sigbody}}{\Gamma \vdash x{:}\mathfrak{S}; S :_c \mathsf{sigbody}}$$

$$\mathsf{MT/SIG}\frac{\Gamma \vdash S :_c \mathsf{sigbody} \qquad \mathrm{Alldiff}(S)}{\Gamma \vdash \mathtt{sig}\ S\ \mathtt{end} :_c \mathsf{modtype}}$$

$$\mathsf{MT/PROD}\frac{\Gamma \vdash M :_c \mathsf{modtype} \qquad \Gamma; x : M \vdash M' :_c \mathsf{modtype}}{\Gamma \vdash \mathtt{funcT}\ x{:}M \to M' :_c \mathsf{modtype}}$$

Specifications

$$\mathsf{WFS/ABSTERM}\frac{\Gamma \vdash t :_w \sigma}{\Gamma \vdash t :_c \mathsf{wfs}}$$

$$\mathsf{WFS/MANTERM}\frac{\Gamma \vdash t :_c t'}{\Gamma \vdash t' :=t :_c \mathsf{wfs}}$$

$$\mathsf{WFS/ABSMOD}\frac{\Gamma \vdash M :_c \mathsf{modtype}}{\Gamma \vdash M :_c \mathsf{wfs}}$$

$$\mathsf{WFS/MANMOD}\frac{\Gamma \vdash m :_c M}{\Gamma \vdash M :=m :_c \mathsf{wfs}}$$

Fig. A 1. Checking module types and specifications.

B Code of the examples

In this appendix, we complete the examples given in our informal presentation, section 3.

In section B.1, we show the content of the current **oeuf** library. Then, in section B.2 we give the proof chunks missing from the informal presentation.

B.1 Standard Library

For the moment, **oeuf** has a minimal standard library. It contains the following declarations packaged in an `Std.iv` interface file:

⟨*standard declarations*⟩≡

```
∨ : Prop → Prop → Prop;
or_introl : ∀ p1 p2 : Prop . p1 → p1 ∨ p2;
or_intror : ∀ p1 p2 : Prop . p2 → p1 ∨ p2;
or_elim : ∀ p1 p2 p : Prop .
  (p1 → p) → (p2 → p) → p1 ∨ p2 → p;
∧ : Prop → Prop → Prop;
and_intro : ∀ p1 p2 : Prop . p1 → p2 → p1 ∧ p2;
and_sym : ∀ p1 p2 : Prop . p1 ∧ p2 → p2 ∧ p1;
fst : ∀ p1 p2 : Prop . p1 ∧ p2 → p1;
snd : ∀ p1 p2 : Prop . p1 ∧ p2 → p2;
```

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

Subtyping

$$\textsc{sub/prod}\ \frac{\Gamma \vdash M_2 <:_c M_1 \qquad \Gamma; x_1 : M_2 \vdash M_1' <:_c M_2'}{\Gamma \vdash \texttt{funcT}\ x_1 : M_1 \to M_1' <:_c \texttt{funcT}\ x_2 : M_2 \to M_2'}$$

$$\textsc{sub/sig}\ \frac{\Gamma; S_1|_{Dom(S_1)} \vdash_c S_2}{\Gamma \vdash \texttt{sig}\ S_1\ \texttt{end} <:_c \texttt{sig}\ S_2\ \texttt{end}}$$

$$\textsc{ent/decl}\ \frac{x \in dom \qquad \Gamma \vdash x :_c \mathfrak{S} \qquad \Gamma|_{dom} \vdash_c S}{\Gamma|_{dom} \vdash_c x : \mathfrak{S}; S}$$

$$\textsc{ent/empty}\ \frac{}{\Gamma|_{dom} \vdash_c \epsilon}$$

Subspecing

$$\textsc{subspec/manm}\ \frac{\Gamma \vdash M <:_c M' \qquad \Gamma \vdash m \bowtie m'}{\Gamma \vdash M :\!= m <:_c M' :\!= m'}$$

$$\textsc{subspec/absm}\ \frac{\Gamma \vdash ty(\mathfrak{S}) <:_c M}{\Gamma \vdash \mathfrak{S} <:_c M}$$

$$\textsc{subspec/mant}\ \frac{\Gamma \vdash t_1 \bowtie t_1' \qquad \Gamma \vdash t_2 \bowtie t_2'}{\Gamma \vdash t_1 :\!= t_2 <:_c t_1' :\!= t_2'}$$

$$\textsc{subspec/abst}\ \frac{\Gamma \vdash ty(\mathfrak{S}) \bowtie t}{\Gamma \vdash \mathfrak{S} <:_c t}$$

Fig. A 2. Checking subtyping and subspecing.

```
nat : Set;
zero : nat;
succ : nat → nat;
le_nat : nat → nat → Prop;
leq_refl : ∀x : nat . x ‘le_nat‘ x;
leq_trans : ∀x y z : nat .
  x ‘le_nat‘ y → y ‘le_nat‘ z → x ‘le_nat‘ z;
s_leq : ∀ x y : nat .
  x ‘le_nat‘ y → (succ x) ‘le_nat‘ (succ y);
```

B.2 Proof chunks not given in the informal presentation

Reflexivity and transitivity of `le_nat` are in fact standard library results:

⟨*proof of reflexivity of* ⩽⟩≡
```
Std.leq_refl
```

⟨*proof of transitivity of* ⩽⟩≡
```
Std.leq_trans
```

⟨*proof that for all* `x,y` *in* `(greatest s)`, `x ⩽ y`⟩≡
```
λ s : a_subset . λ x y : P.a .
 λ maxx : (x ‘elt‘ (greatest s)) .
```

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

Module expressions

$$\mathsf{M/STRUCT}\frac{\Gamma \vdash s ::_{qp} S \qquad \mathrm{Alldiff}(S)}{\Gamma \vdash \texttt{struct}\ s\ \texttt{end} :_{qp} \texttt{sig}\ S\ \texttt{end}}$$

$$\mathsf{M/APP}\frac{\Gamma \vdash m_1 :_{qp} \texttt{funcT}\ x{:}M \to M' \qquad \Gamma \vdash m_2 :_c M}{\Gamma \vdash (m_1\ m_2) :_{qp} M'\{x \leftarrow m_2\}}$$

$$\mathsf{M/LAM}\frac{\Gamma \vdash M :_c \texttt{modtype} \qquad \Gamma; x : M \vdash m :_{qp} M'}{\Gamma \vdash \texttt{functor}\ x{:}M \to m :_{qp} \texttt{funcT}\ x{:}M \to M'}$$

$$\mathsf{M/PATH}\frac{\Gamma \vdash p :_{qp} \mathfrak{S}}{\Gamma \vdash p :_{qp} ty(\mathfrak{S})}$$

Structure bodies

$$\mathsf{S/EMPTY}\frac{}{\Gamma \vdash \epsilon ::_{qp} \epsilon} \qquad \mathsf{S/DECL}\frac{\Gamma \vdash d ::_{qp} \mathfrak{S} \qquad \Gamma\ ;\ x : \mathfrak{S} \vdash s ::_{qp} S}{\Gamma \vdash x{:=}d; s ::_{qp} x{:}\mathfrak{S}; S}$$

Definitions

$$\mathsf{DEF/M1}\frac{\Gamma \vdash m :_{\mathscr{P}} M}{\Gamma \vdash m ::_{qp} M} \qquad \mathsf{DEF/M2}\frac{\Gamma \vdash m :_c M}{\Gamma \vdash m{:}M ::_{qp} M}$$

$$\mathsf{DEF/TRANSPM}\frac{\Gamma \vdash m :_{\mathscr{P}} M}{\Gamma \vdash \texttt{transparent}\ m ::_{qp} M{:=}m}$$

$$\mathsf{DEF/TERM}\frac{\Gamma \vdash t_1 :_{\mathscr{P}} t_2}{\Gamma \vdash t_1 ::_{qp} t_2{:=}t_1}$$

Paths

$$\mathsf{SPEC/VAR}\frac{}{\Gamma \vdash x :_{qp} \Gamma(x)}$$

$$\mathsf{SPEC/SELECT}\frac{\Gamma \vdash m :_{qp} \texttt{sig}\ S\ \texttt{end} \qquad \mathrm{field}(m, x, S) = \mathfrak{S}}{\Gamma \vdash m.x :_{qp} \mathfrak{S}}$$

Fig. A 3. Rules for quasi-principal typing.

```
λ maxy : (y ‘elt‘ (greatest s)) .
 (* y_upper_bound : ∀ z : P.a . z ‘elt‘ s → z ⩽ y *)
 let y_upper_bound :=
   Std.snd (y ‘elt‘ s) (y ‘elt‘ (upper_bounds s)) maxy
 in
 (* x_in_s : x ‘elt‘ s *)
 let x_in_s :=
   Std.fst (x ‘elt‘ s) (x ‘elt‘ (upper_bounds s)) maxx
 in
 y_upper_bound x x_in_s
```

Symmetrically:

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

Principal type of a module

$$\mathsf{M/STR}\frac{\Gamma \vdash m :_{qp} M}{\Gamma \vdash m :_{\mathscr{p}} M/m}$$

Principal specification of a path

$$\mathsf{SPEC/STR}\frac{\Gamma \vdash p :_{qp} \mathfrak{S}}{\Gamma \vdash p :_{\mathscr{p}} \mathfrak{S}/p}$$

Checking typing

$$\mathsf{M/SUB}\frac{\Gamma \vdash m :_{\mathscr{p}} M \qquad \Gamma \vdash M' :_c \mathsf{modtype} \qquad \Gamma \vdash M <:_c M'}{\Gamma \vdash m :_c M'}$$

Checking specification

$$\mathsf{SPEC/SUB}\frac{\Gamma \vdash \mathfrak{S}' :_c \mathsf{wfs} \qquad \Gamma \vdash p :_{\mathscr{p}} \mathfrak{S} \qquad \Gamma \vdash \mathfrak{S} <:_c \mathfrak{S}'}{\Gamma \vdash p :_c \mathfrak{S}'}$$

Fig. A 4. Principal type inference and type checking.

⟨*proof that for all* `x,y` *in* `(lowest s)`, `x ⩽ y`⟩≡

```
λ s : a_subset . λ x y : P.a .
 λ minx : (x ‘elt‘ lowest s) .
 λ miny : (y ‘elt‘ lowest s) .
  (* x_lower_bound : ∀ z : P.a . z ‘elt‘ s → x ⩽ z *)
  let x_lower_bound :=
    Std.snd (x ‘elt‘ s) (x ‘elt‘ (lower_bounds s)) minx
  in
  (* y_in_s : y ‘elt‘ s *)
  let y_in_s :=
    Std.fst (y ‘elt‘ s) (y ‘elt‘ (lower_bounds s)) miny
  in
  x_lower_bound y y_in_s
```

⟨*proof of compatibility of* `succ` *with* ⩽⟩≡

```
Std.s_leq
```

⟨*proof of compatibility of* `f` *with* ⩽⟩≡

```
λx y : Std.nat .
  λh : x ‘Std.le_nat‘ y . Std.leq_refl Std.zero
```

⟨*proof of reflexivity of the intersection* ⩽⟩≡

```
λx : a .
  Std.and_intro (x ‘P1.⩽‘ x) (x ‘P2.⩽‘ x)
    (P1.refl x) (P2.refl x)
```

⟨*proof of transitivity of the intersection* ⩽⟩≡

```
λ x y z : a . λ xy : x ⩽ y . λ yz : y ⩽ z .
   let x1y := Std.fst (x ‘P1.⩽‘ y) (x ‘P2.⩽‘ y) xy in
   let y1z := Std.fst (y ‘P1.⩽‘ z) (y ‘P2.⩽‘ z) yz in
```

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

```
let x1z := P1.trans x y z x1y y1z in
let x2y := Std.snd (x `P1.⩽` y) (x `P2.⩽` y) xy in
let y2z := Std.snd (y `P1.⩽` z) (y `P2.⩽` z) yz in
let x2z := P2.trans x y z x2y y2z in
Std.and_intro (x `P1.⩽` z) (x `P2.⩽` z) x1z x2z
```

C Names and α-conversion

One would be tempted to say that we can deal with renaming and α-conversion the usual way, for instance saying that we perform renaming when substituting and when introducing a variable in a context (in order the introduced variable to be fresh). However renaming raises some problems with structures (as well as with signatures) as they are dependently typed (Harper & Lillibridge, 1994). For instance if m is a structure `struct` x`:=`m'`;`y`:=`$z.w$`;` `end` then what should be $m\{z \leftarrow x\}$? If we replace $z.w$ by $x.w$, as in `struct` x`:=`m'`;`y`:=`$x.w$`;` `end` then x in $x.w$ is captured by the definition x`:=`m'. On the other hand, we cannot rename the field x of m as field names are meaningful (whereas names of lambda-abstraction can be freely renamed). In order to solve this problem, Harper and Lillibridge noticed that x had two roles in the signatures: it is both a name which helps accessing a field from the outside and a binder for fields that follow it in the signature declaration.

Therefore, adopting their solution, we distinguish these roles. A field declaration in a structure or a signature has shape x `as` y `:` …, the former identifier being a name, and the latter being a binder, whose name is subject to renaming. For instance, the following three signatures are α-equivalent:

```
sig
 a as a: Set;
 r as r: a -> a -> Prop;
end
```

```
sig
 a as a': Set;
 r as r': a' -> a' -> Prop;
end
```

```
sig
 a as a'': Set;
 r as r'': a'' -> a'' -> Prop;
end
```

while in the following signature, the variable `a` is free:

```
sig
 a as a': Set;
 r as r': a -> a -> Prop;
end
```

In fact, the notation x `:` … we used in this paper was just syntactic sugar for x `as` x `:` ….

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

Note that the only rule that really needs to deal explicitly with this distinction is the rule [ENT/DECL] described in 6.1.

Finally, as we work with De Bruijn indices, the only rule that needs a special treatment with respect to renaming is [SUB/SIG] described in 6.1, let alone the usual lifting operations (the function looking for a declaration in the environment much appropriately lift the specification it returns and the substitution function must also apply the adequate lifting operations).

D Untyped $\beta_m\rho\delta$-reduction is not Church-Rosser

In lambda-calculi with dependent types or higher-order types, convertibility is most often defined as the congruent closure of β-reduction on untyped terms. This notion of convertibility can then be proved equivalent to the equality of β-normal forms. Since the work of Hermann Geuvers (Geuvers, 1993), it is well known that things do not go that smooth with $\beta\eta$-reduction: the untyped congruent closure of $\beta\eta$-reduction is not equivalent to the equality of $\beta\eta$-normal form on untyped terms as the $\beta\eta$-reduction does not enjoy the Church-Rosser on untyped terms. However, the divergence is quite limited as two terms obtained by $\beta\eta$-reduction can reduce to two terms differing only by the types labeling binders in lambda-abstractions. Geuvers used this fact to prove that $\beta\eta$-reduction is Church-Rosser on typed terms and that the untyped congruent closure of $\beta\eta$-reduction is equivalent to the relation "having a common reduct by $\beta\eta$" on typed terms.

However, Geuvers' approach can not apply to $\mathscr{MC}_2$, as $\triangleright$ in $\mathscr{MC}_2$ diverges much more, as the following lemma shows.

Lemma D.1
Given an environment Γ and two base terms (resp. two module expressions) q_1 and q_2, there exists a base term (resp. a module expression) q such that $\Gamma \vdash q \triangleright^\star q_1$ and $\Gamma \vdash q \triangleright^\star q_2$.

Proof
Let q_1 and q_2 be two modules expressions (resp. two base terms) and τ be a module type (resp. a base term). Let

$$\begin{aligned} M_2 &= \texttt{sig X: } \tau \texttt{:=} q_2 \texttt{; end} \\ m_1 &= \texttt{struct X:=} q_1 \texttt{; end} \\ m_2 &= \texttt{struct Y:=} q_2 \texttt{; end} \\ m &= \texttt{functor Z:} M_2 \rightarrow \texttt{struct Y:=Z.X; end} \\ q &= (m\ m_1)\texttt{.Y} \end{aligned}$$

q reduces (in many steps) to both q_1 and q_2. Indeed, we have

$$\begin{aligned} q &\triangleright_{\beta_m} \texttt{struct Y:=}m_1\texttt{.X; end.Y} \\ &\triangleright_{\rho} m_1\texttt{.X} \\ &\triangleright_{\rho} q_1 \end{aligned}$$

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

and also

$$\begin{array}{lll} q & \triangleright_{\delta} & ((\texttt{functor Z}\!:\!M_2 \to m_2)\ m_1)\,.\,\texttt{Y} \\ & \triangleright_{\beta_m} & m_2\,.\,\texttt{Y} \\ & \triangleright_{\rho} & q_2 \end{array}$$

□

Proposition D.2
The reflexive transitive closure of ▷ is the full relation.

Proof
This is a direct corollary of the previous lemma. □

Therefore, one has to consider the reflexive transitive closure of ▷ restricted to typed terms and typed modules. As dealing with the metatheory of this relation is difficult, we choose to consider instead the equality of $\beta\beta_m\rho\delta$ normal forms instead. Once the metatheory is done, it is easy to show both of them are equivalent, thanks to subject-reduction, Church-Rosser and normalization.

References

Ancona, Davide, & Zucca, Elena. (1996). An algebraic approach to mixins and modularity. *Pages 179–193 of:* Hanus, M., & Artalejo, M. Rodrìguez (eds), *5th intl. conf. on algebraic and logic programming*. Lecture Notes in Computer Science, no. 1139. Berlin: Springer-Verlag.

Aponte, M.-V., & Castagna, G. (1996). Programmation modulaire avec surcharge et liaison tardive. *Journées francophones des langages applicatifs.*

Asperti, Andrea. 1999 (Nov.). *History, advancements, and projectual choices.* Web page. `http://www.cs.unibo.it/~asperti/HELM/progress.html`. See the *November 26* entry.

Augustsson, Lennart. (1998). Cayenne – a language with dependent types. *Pages 239–250 of: International conference on functional programming.*

Augustsson, Lennart. (1999). *Cayenne — Hotter than Haskell.* `http://www.cs.chalmers.se/~augustss/cayenne/`.

Barendregt, Henk. (1993). Typed lambda calculi. Abramsky et al. (ed), *Handbook of logic in computer science*. Oxford Univ. Press.

Barras, B., Boutin, S., Cornes, C., Courant, J., Coscoy, Y., Delahaye, D., de Rauglaudre, D., Filliâtre, J.C., Giménez, E., Herbelin, H., Huet, G., Laulhère, H., Loiseleur, P., Muñoz, C., Murthy, C., Parent, C., Paulin, C., Saïbi, A., & Werner, B. 1999 (July). *The Coq Proof Assistant Reference Manual – Version V6.3.* `http://coq.inria.fr/doc/main.html`.

Bertot, Yves, Grgoire, Benjamin, & Leroy, Xavier. (2005). A structured approach to proving compiler optimizations based on dataflow analysis. *Types for proofs and programs, workshop types 2004.* Lecture Notes in Computer Science. Springer-Verlag.

Betarte, Gustavo. (1998). *Dependent record types and algebraic structures in type theory.* Ph.D. thesis, PMG, Dept. of Computing Science, University of Göteborg and Chalmers University of Technology.

Betarte, Gustavo. 2000a (Sept.). *Proof reutilization in Martin-Löf's logical framework extended with record types and subtyping.* Tech. rept. 00-09. InCo-Pedeciba.

Betarte, Gustavo. (2000b). Type checking dependent (record) types and subtyping. *Journal of functional programming*, **10**(2), 137–166.

Bourbaki, Nicolas. (1970). *Eléments de mathématique; théorie des ensembles*. Paris: Hermann. Chap. IV.

Cachera, David, Jensen, Thomas, Pichardie, David, & Schneider, Gerardo. (2005). Certified Memory Usage Analysis. *Procòf 13th international symposium on formal methods (fm'05)*. Lecture Notes in Computer Science. `http://www.springer.de/comp/lncs/index.html`Springer-Verlag. to appear.

Cardelli, Luca. (1997). Program fragments, linking, and modularization. *Pages 266–277 of: Conference record of the 24th symposium on principles of programming languages*. Paris, France: ACM Press.

Chrząszcz, Jacek. 2003 (sep). Implementation of modules in the coq system. *Pages 270–286 of:* Basin, David, & Wolff, Burkhart (eds), *Theorem proving in higher order logic*. LNCS, vol. 2758. Roma, Italy.

Chrząszcz, Jacek. (2004a). Modules in coq are and will be correct. *Pages 130–146 of:* Berardi, Stefano, Coppo, Mario, & Damiani, Ferruccio (eds), *Types for proofs and programs, international workshop, types 2003*. LNCS, vol. 3085. Springer-Verlag. Revised Selected Papers.

Chrząszcz, Jacek. (2004b). *Modules in type theory with generative definitions*. Ph.D. thesis, Université Paris-Sud.

Coq. *The Coq Proof Assistant*. `http://coq.inria.fr/`.

Coquand, C., & Coquand, T. (1999). Structured type theory. *Proc. workshop on logical frameworks and meta-languages (lfm'99)*.

Coquand, Catarina. (2000). *Agda*. `http://www.cs.chalmers.se/~catarina/agda/`.

Coquand, T., & Gallier, J. H. (1990). A proof of strong normalization for the Theory of Constructions using a Kripke-like interpretation. *Pages 479–497 of:* Huet, G., & Plotkin, G. (eds), *Preliminary proceedings 1st intl. workshop on logical frameworks, antibes, france, 7–11 may 1990*.

Courant, Judicaël. (1997). An applicative module calculus. *Pages 622–636 of: Theory and practice of software development 97*. Lecture Notes in Computer Science. Lille, France: Springer-Verlag.

Courant, Judicaël. (1998). *Un calcul de modules pour les systèmes de types purs*. Thèse de doctorat, Ecole Normale Supérieure de Lyon.

Courant, Judicaël. 1999 (June). *$\mathcal{MC}$: A module calculus for Pure Type Systems*. Research Report 1217. LRI.

Courant, Judicaël. (2002a). Explicit universes for the calculus of constructions. *Pages 115–130 of:* Carreño, Victor A., Muñoz, César A., & Tahar, Sofiène (eds), *Theorem proving in higher order logics: 15th international conference, tphols 2002*. Lecture Notes in Computer Science, vol. 2410. Hampton, VA, USA: Springer-Verlag.

Courant, Judicaël. (2002b). Strong normalization with singleton types. Bakel, Stefan Van (ed), *Second workshop on intersection types and related systems*. Electronic Notes in Computer Science, vol. 70. Copenhaguen, Danemark: Elsevier Science BV.

Crary, Karl, Harper, Robert, & Puri, Sidd. (1999). What is a recursive module? *Pages 50–63 of: SIGPLAN conference on programming language design and implementation*.

Dreyer, Derek. 2002 (Sept.). *Moscow ml's higher-order modules are unsound*. Post on the Types Forum. Available at `http://www.cis.upenn.edu/\home{bcpierce}/types/archives/current/msg0113%6.html`.

Dreyer, Derek, Crary, Karl, & Harper, Robert. 2002 (July). *A type system for higher-order modules*. Technical Report CMU-CS-02-122. CMU.

Farmer, W. M., Guttman, J. D., & Thayer Fábrega, F. J. (1996). IMPS: An updated system description. *Pages 298–302 of:* McRobbie, M. A., & Slaney, J. K. (eds), *13th international*

conference on automated deduction. Lecture Notes in Computer Science. New Brunswick, NJ: Springer-Verlag.

Farmer, William M. (2000). An infrastructure for intertheory reasoning. *Pages 115–131 of:* McAllester, D. (ed), *17th international conference on automated deduction*. Lecture Notes in Computer Science, vol. 1831. Pittsburgh, PA, USA: Springer-Verlag.

Farmer, William M., Guttman, Joshua D., & Thayer, F. Javier. (1992). Little theories. *Pages 567–581 of:* Kapur, Deepak (ed), *11th international conference on automated deduction*. Lecture Notes in Computer Science, vol. 607. Saratoga Springs, NY: Springer-Verlag.

Farmer, William M., Guttman, Joshua D., & Thayer, F. Javier. 1995 (Apr.). *The* IMPS *user's manual*. The MITRE Corporation, Bedford, MA 01730 USA. Available from `http://imps.mcmaster.ca/`.

Filliâtre, J.-C., & Letouzey, P. 2004 (April). Functors for Proofs and Programs. *Pages 370–384 of: Proceedings of the european symposium on programming*. Lecture Notes in Computer Science, vol. 2986.

Flatt, Matthew, & Felleisen, Matthias. (1998). Units: Cool modules for HOT languages. *Pages 236–248 of: Proceedings of the ACM SIGPLAN '98 conference on programming language design and implementation*.

Geuvers, Herman. 1993 (Sept.). *Logics and type systems*. Ph.D. thesis, University of Nijmegen.

Goguen, Healfdene. 1994 (Aug). *A typed operational semantics for type theory*. Ph.D. thesis, University of Edinburgh. LFCS Report ECS-LFCS-94-304.

Hallgren, Thomas. *Alfa web page*. `http://www.cs.chalmers.se/~hallgren/Alfa/`.

Harper, R., & Lillibridge, M. (1994). A type-theoretic approach to higher-order modules with sharing. *In:* (POPL'94, 1994).

Harper, R., & Pollack, R. (1991). Type checking, universe polymorphism, and typical ambiguity in the calculus of constructions. *Theoretical computer science*, **89**(1).

Harper, R., Mitchell, J., & Moggi, E. (1990). Higher-order modules and the phase distinction. *Pages 341–354 of: 17th symposium on principles of programming languages*. ACM.

Harper, Robert. 2002 (Mar.). *Programming in standard ML*. Available from `http://www-2.cs.cmu.edu/~rwh/smlbook/`.

Harper, Robert, & Pfenning, Frank. 1992 (Sept.). *A module system for a programming language based on the LF logical framework*. Tech. rept. CMU-CS-92-191. Carnegie Mellon University, Pittsburgh, Pennsylvania.

Harper, Robert, & Pfenning, Frank. (1998). A module system for a programming language based on the LF logical framework. *Journal of logic and computation*, **1**(8).

Hirschowitz, Tom, & Leroy, Xavier. 2002 (Apr.). Mixin modules in a call-by-value setting. *Pages 6–20 of:* Le Mtayer, Daniel (ed), *ESOP*. Lecture Notes in Computer Science, vol. 2305.

Johnson, Andrew L., & Johnson, Brad C. (1997). Literate programming using noweb. *Linux journal*, Oct., 64–69.

Jones, Alex, Luo, Zhaohui, & Soloviev, Sergei. (1998). Some Algorithmic and Proof-Theoretical Aspects of Coercive Subtyping. *Proceedings of TYPES'96*. Lecture Notes in Computer Science.

Kammüller, Florian. (1996). *Comparison of IMPS, PVS and Larch with respect to theory treatment and modularization*. Unpublished draft available from `http://www.first.gmd.de/~florian/papers/index.html`.

Kammüller, Florian. (2000). Modular reasoning in isabelle. *17th international conference on automated deduction, cade-17*. Lecture Notes in Artificial Intelligence, vol. 1831. Springer-Verlag.

Kammüller, Florian, Wenzel, Markus, & Paulson, Lawrence C. (1999). Locales – a sectionning concept for isabelle. Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., & Thery, L. (eds), *Theorem proving in higher order logics, 12th international conference, tphols'99*. Lecture Notes in Computer Science, vol. 1657. Springer-Verlag. Available from `http://www.first.gmd.de/~florian/papers/`.

Lego. *The LEGO proof assistant*. `http://www.dcs.ed.ac.uk/home/lego/`.

Leroy, Xavier. (1994). Manifest types, modules, and separate compilation. *In:* (POPL'94, 1994).

Leroy, Xavier. (1995). Applicative functors and fully transparent higher-order modules. *Pages 142–153 of: Conference record of the 22nd symposium on principles of programming languages*. San Francisco, California: ACM Press.

Leroy, Xavier, Doligez, Damien, Garrigue, Jacques, Rémy, Didier, & Vouillon, Jérôme. 2001 (Mar.). *The Objective Caml system release 3.01, documentation and user's manual*. `http://www.caml.org/ocaml/htmlman/index.html`.

Lillibridge, Mark. 1997 (May). *Translucent Sums: A Foundation for Higher-Order Module Systems*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University.

Luo, Zhaohui. (1990). *An extended calculus of constructions*. Ph.D. thesis, University of Edinburgh.

MacQueen, David B. (1984). Modules for Standard ML. *Pages 198–207 of: Proceedings of the 1984 ACM conference on LISP and functional programming*. ACM Press.

Milner, Robin, Tofte, Mads, Harper, Robert, & MacQueen, David. (1997). *The definition of standard ml (revised)*. MIT Press.

Owre, S., Shankar, N., Rushby, J. M., & Stringer-Calvert, D. W. J. 1999 (Sept.). *Pvs language reference*. Computer Science Laboratory, SRI International, Menlo Park, CA.

Owre, Sam, & Shankar, Natarajan. 1997 (Aug.). *The formal semantics of PVS*. Tech. rept. SRI-CSL-97-2. Computer Science Laboratory, SRI International, Menlo Park, CA.

Pollack, Randy. 1997 (Aug.). *Theories in Type Theory*. Talk at the Types Working Group Workshop on Subtyping, Inheritance and Modular Development of Proofs. Draft available at `http://www.dcs.ed.ac.uk/home/rap/export/`.

Pollack, Randy. 2000 (Aug.). *Dependently Typed Records for Representing Mathematical Structures*. Available at `http://www.dcs.ed.ac.uk/home/rap/export/records.ps.gz`. A previous version of this paper appears in *Theorem Proving in Higher Order Logics (TPHOLs 2000)*.

POPL'94. (1994). *Conference record of the 21st symposium on principles of programming languages*. Portland, Oregon: ACM Press.

Ramsey, Norman. (1994). Literate programming simplified. *Ieee software*, **11**(5), 97–105.

Ramsey, Norman. (2001). *Noweb — A Simple, Extensible Tool for Literate Programming*. `http://www.eecs.harvard.edu/~nr/noweb/`.

Russo, Claudio V. (1998). *Types for modules*. LFCS thesis ECS-LFCS-98-389, University of Edinburgh.

Severi, Paula. (1996). *Normalisation in lambda calculus and its relation to type inference*. Ph.D. thesis, Eindhoven University of Technology.

Shankar, N., Owre, S., & Rushby, J. M. 1993 (Feb.). *Pvs tutorial*. Computer Science Laboratory, SRI International, Menlo Park, CA. Also appears in Tutorial Notes, *Formal Methods Europe '93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993.

Stone, Christopher A., & Harper, Robert. (2000). Deciding type equivalence in a language with singleton kinds. *Pages 214–227 of:* Reps, Thomas (ed), *Conference record of the 27th symposium on principles of programming languages*. Boston, Masschusetts: ACM Press. Available at `http://www.cs.hmc.edu/~stone/papers/popl00-preprint.ps`.

https://doi.org/10.1017/S0956796806005867 Published online by Cambridge University Press

Tasistro, Alvaro. (1997). *Substitution, record types and subtyping in type theory, with applications to the theory of programming*. Ph.D. thesis, PMG, Dept. of Computing Science, University of Göteborg and Chalmers University of Technology.

Tofte, Mads. (1996). Essentials of Standard ML modules. *Pages 208–238 of:* Launchbury, John, Meijer, Eric, & Sheard, Tim (eds), *Advanced functional programming*. Lecture Notes in Computer Science, vol. 1129. Springer-Verlag.

Twelf. *The Twelf project*. `http://www.cs.cmu.edu/~twelf/`.

van Benthem Jutting, L. S., McKinna, J., & Pollack, R. 1993 (May). Checking algorithms for pure type systems. *Types for proofs and programs: International workshop types'93*. Lecture Notes in Computer Science, vol. 806.

Wenzel, Markus. 2001 (Feb.). *Using axiomatic type classes in isabelle*. TU München. Available from Isabelle web site.