Theories as Types

Dennis Müller[1], Florian Rabe[1,2], and Michael Kohlhase[1]

[1] Computer Science, FAU Erlangen-Nürnberg
[2] LRI, Université Paris Sud

Abstract. Theories are an essential structuring principle that enable modularity, encapsulation, and reuse in formal libraries and programs (called classes there). Similar effects can be achieved by dependent record types. While the former form a separate language layer, the latter are a normal part of the type theory. This overlap in functionality can render different systems non-interoperable and lead to duplication of work.

We present a type-theoretic calculus and implementation of a variant of record types that for a wide class of formal languages naturally corresponds to theories. Moreover, we can now elegantly obtain a contravariant functor that reflects the theory level into the object level: for each theory we obtain the type of its models and for every theory morphism a function between the corresponding types. In particular this allows shallow – and thus structure-preserving – encodings of mathematical knowledge and program specifications while allowing the use of object-level features on models, e.g. equality and quantification.

1 Introduction

In the area of formal systems like type theories, logics, and specification and programming languages, various language features have been studied that allow for inheritance and modularity, e.g., theories, classes, contexts, and records. They all share the motivation of grouping a list of declarations into a new entity such as in $R = [\![x_1 : A_1, \ldots, x_n : A_n]\!]$. The basic intuition behind it is that R behaves like a product type whose values are of the form $\langle x_1 : A_1 := a_1, \ldots, x_n : A_n := a_n \rangle$. Such constructs are indispensable already for elementary applications such as defining the algebraic structure of Semilattices (as in Figure 1), which we will use as a running example.

$$\texttt{Semilattice} = \left\{ \begin{array}{ll} U & : \texttt{type} \\ \wedge & : U \to U \to U \\ \texttt{assoc} & :\vdash \forall x, y, z : U.\ (x \wedge y) \wedge z \doteq x \wedge (y \wedge z) \\ \texttt{commutative} & : \ldots \\ \texttt{idempotent} & : \ldots \end{array} \right\}$$

Fig. 1. A Grouping of Declarations for Semilattices

Many systems support **stratified grouping** (where the language is divided into a lower level for the base language and a higher level that introduces the grouping constructs) or **integrated grouping** (where the grouping construct is one out of many type-forming operations without distinguished ontological status), or both. The names of the grouping constructs vary between systems, and we will call them **theories** and **records** in the sequel. An overview of some representative examples is given in the table on the right.

System	Name of feature	
	stratified	integrated
ML	signature/module	record
C++	class	class, struct
Java	class	class
Idris [Bra13]	module	record
Coq [Coq15]	module	record
HOL Light [Har96]	ML signatures	records
Isabelle [Wen09]	theory, locale	record
Mizar [TB85]	article	structure
PVS [ORS92]	theory	record
OBJ [Gog+93]	theory	
FoCaLiZe [Har+12]	species	record

The two approaches have different advantages. Stratified grouping permits a separation of concerns between the core language and the module system. It also captures high-level structure well in a way that is easy to manage and discover in large libraries, closely related to the advantages of the little theories approach [FGT92]. But integrated grouping allows applying base language operations (such as quantification or tactics) to the grouping constructs. For this reason, the (relatively simple) stratified Coq module system is disregarded in favor of records in major developments such as [Mat].

Allowing both features can lead to a duplication of work where the same hierarchy is formalized once using theories and once using records. A compromise solution is common in object-oriented programming languages, where classes behave very much like stratified grouping but are at the same time normal types of the type system. We call this **internalizing** the higher level features. While combining advantages of stratified and integrated grouping, internalizing is a very heavyweight type system feature: stratified grouping does not change the type system at all, and integrated grouping can be easily added to or removed from a type system, but internalization adds a very complex type system feature from the get-go. It has not been applied much to logics and similar formal systems: the only example we are aware of is the FoCaLiZe [Har+12] system. A much weaker form of internalization is used in OBJ and related systems based on stratified grouping: here theories may be used as (and only as) the types of parameters of parametric theories. Most similarly to our approach, OCaml's first-class modules internalize the theory (called *module type* in OCaml) M as the type `module` M; contrary to both OO-languages and our approach, this kind of internalization is in addition and unrelated to integrated grouping.

In any case, because theories usually allow for advanced declarations like imports, definitions, and notations, as well as extra-logical declarations, systematically internalizing theories requires a correspondingly expressive integrated

grouping construct. Records with defined fields are comparatively rare; e.g., present in [Luo09] and OO-languages. Similarly, imports between record types and/or record terms are featured only sporadically, e.g., in Nuprl [Con+86], maybe even as an afterthought only.

Finally, we point out a closely related trade-off that is orthogonal to our development: even after choosing either a theory or a record to define grouping, many systems still offer a choice whether a declaration becomes a parameter or a field. See [SW11] for a discussion.

Contribution We present the first formal system that systematically internalizes theories into record types. The central idea is to use an operator `Mod` that turns the theory T into the type $\texttt{Mod}\,(T)$, which behaves like a record type. We take special care not to naively compute this record type, which would not scale well to the common situations where theories with hundreds of declarations or more are used. Instead, we introduce record types that allow for defined fields and merging so that $\texttt{Mod}\,(T)$ preserves the structure of T.

Our approach combines the advantages of stratified and integrated grouping in a lightweight language feature that is orthogonal to and can be easily combined with other foundational language features. Concretely, it is realized as a module in the Mmt framework [Rab17b], which allows for the modular design of foundational languages. By combining our new modules with existing ones, we obtain many formal systems with internalized theories. In particular, our typing rules conform to the abstractions of Mmt so that Mmt's type reconstruction [Rab17a] is immediately applicable to our features. We showcase the potential in a case study based on this implementation, and which is interesting in its own right: A formal library of elementary mathematical concepts that systematically utilizes $\texttt{Mod}\,(\cdot)$ throughout for algebraic structures, topological spaces etc.

Overview We formulate our approach in the setting of a dependently-typed λ-calculus, which we recall in Sect. 2. This section also serves as a gentle primer for defining language features in Mmt. Sect. 4 introduces our notion of record types, based on which we introduce the model-operator in Sect. 5. Sect. 6 presents our implementation and a major case study on elementary mathematics. This paper is an extended version of [MRK18].

2 Preliminaries

We introduce the well-known dependently-typed lambda calculus as the starting point of our development. The **grammar** is given in Figure 2. The only surprise here is that we allow optional definitions in contexts; this is a harmless convenience at this point but will be critical later on when we introduce records with defined fields. As usual, we write $T \to T'$ instead of $\prod_{x:T} T'$ when possible. We also write $T[x/T']$ for the usual capture-avoiding **substitution** of T' for x in T.

Mmt uses a bidirectional type system, i.e., we have two separate judgments for type *inference* and type *checking*. Similarly, we have two equality judgments:

$\Gamma ::= \cdot \mid \Gamma, x[: T][:= T]$	contexts
$T ::= x \mid \texttt{type} \mid \texttt{kind}$	variables and universes
$\mid \prod_{x:T'} T \mid \lambda x : T'.T \mid T_1 T_2$	dependent function types

Fig. 2. Grammar for Contexts and Expressions

one for checking equality of two given terms and one for reducing a term to another one. Our **judgments** are given in Figure 3.

Adding record types in Section 4 will introduce non-trivial **subtyping**, e.g., $[\![x : T, y : S]\!]$ is a subtype of $[\![x : T]\!]$.[3] Therefore, we already introduce a subtyping judgment here even though it is not needed for dependent function types yet. For our purposes, it is sufficient (and desirable) to consider subtyping to be an abbreviation: $\Gamma \vdash T_1 <: T_2$ iff for all t $\Gamma \vdash t \Leftarrow T_1$ implies $\Gamma \vdash t \Leftarrow T_2$.

Judgment	Intuition
$\vdash \Gamma$ `ctx`	Γ is a well-formed context
$\Gamma \vdash t \Leftarrow T$	t checks against type/kind T.
$\Gamma \vdash t \Rightarrow T$	type/kind of term t is inferred to be T
$\Gamma \vdash t_1 \equiv t_2 : T$	t_1 and t_2 are equal at type T
$\Gamma \vdash t_1 \rightsquigarrow t_2$	t_1 computes to t_2
$\Gamma \vdash T_1 <: T_2$	T_1 is a subtype of T_2

Fig. 3. Judgments

The **pre/postconditions** of these judgments are as follows: $\Gamma \vdash t \Leftarrow T$ assumes that T is well-typed and implies that t is well-typed. $\Gamma \vdash t \Rightarrow T$ implies that both t and T are well-typed. $\Gamma \vdash t_1 \rightsquigarrow t_2$ implies that t_2 is well-typed iff t_1 is (which puts additional burden on computation rules that are called on not-yet-type-checked terms). Equality and subtyping are only used for expressions that are assumed to be well-typed, i.e., $\Gamma \vdash t_1 \equiv t_2 : T$ implies $\Gamma \vdash t_i \Leftarrow T$, and $\Gamma \vdash T_1 <: T_2$ implies that T_i is a type/kind.

Remark 1 (Horizontal Subtyping and Equality). The equality judgment could alternatively be formulated as an untyped equality $t \equiv t'$. That would require some technical changes to the rules but would usually not be a huge difference. In our case, however, the use of typed equality is critical.

For example, consider record values $r_1 = \langle a := 1, b := 1 \rangle$ and $r_2 = \langle a := 1, b := 2 \rangle$ as well as record types $R = [\![a : nat]\!]$ and $S = [\![a : nat, b : nat]\!]$. Due to horizontal subtyping, we have $S <: R$ and thus both $r_i \Leftarrow S$ and $r_i \Leftarrow R$. This has the advantage that the function $S \to R$ that throws away the field b becomes the identity operation. Now our equality at record types behaves accordingly and checks only for the equality of those fields required by the type. Thus, $r_1 \equiv r_2 : R$ is true whereas $r_1 \equiv r_2 : S$ is false, i.e., the equality of two terms may depend on the type at which they are compared. While seemingly dangerous, this makes

[3] This is sometimes called *horizontal* subtyping. In that case, the straightforward covariance rule for record types is called *vertical* subtyping.

sense intuitively: r_1 can be replaced with r_2 in any context that expects an object of type R because in such a context the field b, where r_1 and r_2 differ, is inaccessible.

Of course, this treatment of equality precludes downcasts: an operation that casts the equal terms $r_1 : R$ and $r_2 : R$ into the corresponding unequal terms of type S would be inconsistent. But such downcasts are still possible (and valuable) at the meta-level. For example, a tactic $GroupSimp(G, x)$ that simplifies terms x in a group G can check if G is commutative and in that case apply more simplification operations.

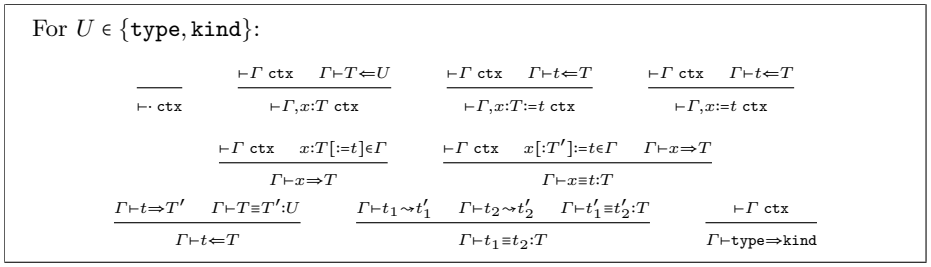

Fig. 4. General Rules

The general **rules** of the framework are given in Figure 4. The upper row contains the rules for contexts. Note that even though we allow the type T of a variable to be omitted (which will be helpful for records later), that is only allowed if a definiens t is present. (T must be inferable from t or otherwise known from the environment.) The middle row contains the rules for looking up the type and definition of variable. The bottom row contains the bidirectionality rules, which algorithmically are the default rules that are applied when no type (resp. equality) checking rules are available: switch to type inference (resp. computation) and compare inferred and expected type (resp. the results). The last rule introduces the two universes. We refer to [Rab17a] for the general rules about equality, which we omit here. They consist of α-renaming and the rules that make equality a congruence relation.

The specific rules for the dependent function types are given in Figure 5. These rules follow a general pattern: The upper row contains one inference rule for each constructor. The bottom row contains a type checking rule and an equality checking rule (i.e., extensionality) at $\prod$-types as well as the usual β-computation rule. We do not have an η-rule because it is equivalent to extensionality.

We can now show that the usual variance rule for function types is derivable

Theorem 1. *The following subtyping rule is derivable:*

$$\frac{\Gamma \vdash A <: A' \quad \Gamma, x : A \vdash B' <: B}{\Gamma \vdash \prod_{x:A'} B' <: \prod_{x:A} B}$$

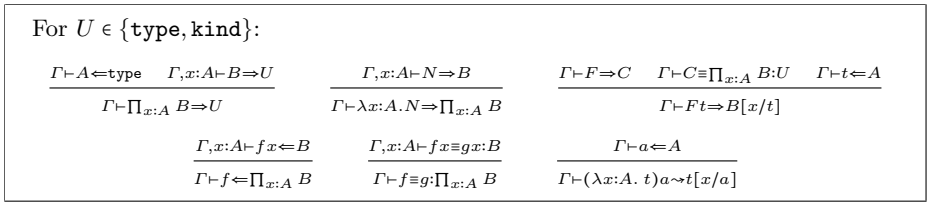

Fig. 5. Rules for Dependent Function Types

Proof. Assume $\Gamma \vdash A <: A'$, $\Gamma \vdash B' <: B$. We need to show $\Gamma, f : \prod_{x:A'} B' \vdash f \Leftarrow \prod_{x:A} B$.

By Figure 5 we need to show $\Gamma, f : \prod_{x:A'} B', x : A \vdash f\, x \Leftarrow B$. Since $\Gamma \vdash A <: A'$ we have $\Gamma, x : A \vdash x \Leftarrow A'$ and consequently $\Gamma, f : \prod_{x:A'} B', x : A \vdash f\, x \Leftarrow B'[x/x]$, so the claim follows by $\Gamma \vdash B' <: B$.

Moreover, we can show that every well-typed term t has a **principal type** T in the sense that (i) $\Gamma \vdash t \Leftarrow T$ and (ii) whenever $\Gamma \vdash t \Leftarrow T'$, then also $\Gamma \vdash T <: T'$. The principal type is exactly the one inferred by our rules (see Theorem 2).

3 Related Work

Languages can differ substantially in the syntax and semantics of these constructs. Our interest here is in one difference in particular, which we call the difference between **stratified** and **integrated** grouping.

3.1 Analysis

With **stratified** grouping, the language is divided into a lower level for the base language and a higher level that introduces the grouping constructs. For example, the SML module system is stratified: it uses a simply typed λ-calculus at the lower level and *signatures* for the type-like and *structures* for the value-like grouping constructs at the higher level. Critically, the higher level constructs are not valid objects at the lower level: even though signatures behave similarly to types, they are not types of base language. With **integrated** grouping, only one level exists: the grouping construct is one out of many type-forming operations of the base language with no distinguished ontological status. For example, SML also provides record types as a grouping construct that is integrated with the type system.

Stratified languages have the advantage that they can be designed in a way that yields a conservativity property: all higher level features can be seen as abbreviations that can be compiled into base language. This corresponds to a typical historical progression where a simple base language is designed first and studied theoretically (e.g., the simply-typed λ-calculus) and grouping is added later when practical applications demand it. But they have the disadvantage that they tend towards a duplication of features: many operations of the lower level

are also desirable at the higher level. For example, SML *functors* are essentially functions whose domain and codomain are signatures, a duplication of the function types that already exist in the base language. In logics, this problem is even more severe because quantification and equality (and eventually tactics, decision procedures etc.) quickly become desirable at the higher level as well, at which point a duplication of features tends to become infeasible. A well-known example of this trap is the stratified Coq module system (inspired by SML), which practitioners often dismiss in favor of using record types, most importantly in the mathematical components project [Mat].

This may lead us to believe that record types are the way to go — but this is not ideal either. Record types usually do not support advanced declarations like imports, definitions, and notations, which are commonplace in stratified languages and indispensable in practice. Depending on the system, record types may also forbid some declarations such as type declarations (which would require a higher universe to hold the record type), dependencies between declarations (which would require dependent types), and axioms (which do not fit the record paradigm in systems that do not use a propositions-as-types design). And complex definition principles such as for inductive types and recursive functions are often placed into a stratified higher level just to handle their inherent difficulty. Moreover, stratified grouping has proved very appropriate for organizing extra-logical declarations such as prover instructions (e.g., tactics, rewrite rules, unification hints) examples, sectioning, comments, and metadata. While some systems use files as a simple, implicit higher level grouping construct, most systems use an explicit one. The exalted status of higher level grouping also often supports documentation and readability because it makes the large-scale structure of a development explicit and obvious. This is particularly helpful when formalizing software specifications or mathematical theories, whose structure naturally corresponds to those offered by higher-level grouping. In their work on integrating theorem prover libraries, the authors have experienced that this correspondence makes it much easier to compare and integrate different stratified formalizations of the same concepts.

3.2 Related Work

[Luo09] presents an extension of the logical framework LF with record types with *manifest fields*. The latter are implemented using unit types and implicit type coercions are used for prjecting the manifest fields. The main difference is that our implementation uses the same contexts for both stratified and integrated groupings, which allows for bridging the gap between the two easily. Furthermore, we allow *merging* of records and record types (see Section 4.2) to make manifest fields accessible without having to restate them for each instance of a type.

OO-languages use classes for stratified groupings, the corresponding types of which basically represent integrated groupings. Classes however can usually only ever linearly extend one superclass and need all of their fields (except for parameters) instantiated.

Interfaces (e.g. in Java) somewhat fix the latter problem by allowing almost arbitrary mixing, but have different restrictions in that fields may not be defined in interfaces at all.

Traits and *Abstract Classes* in Scala finally allow almost arbitrary mixing and implementing or omitting of definitions. These probably come closest to our implementation.

PVS [ORS92] is an interactive theorem prover system with a highly expressive language. PVS uses *theories* (which may be parametric) for stratified groupings and records for integrated groupings, but no mechanism to convert between the two exists.

While theories may extends arbitrary other theories, record types an only be extended linearly and only by fields *independent* of the base type they extend – so basically only two record types that are independently well-typed can be merged.

Agda [Nor05] uses Modules (which may be parametric) for stratified groupings and records for integrated groupings. While all "fields" in a Module need to be defined, the parameters can be thought of as undefined fields in a record type. Parameters are abstracted away when a module is closed.

Additionally, Agda has records. While there is no primitive way to convert modules to record types, converting records to modules manually to open all fields in the current context seems to be a popular trick.

Coq [Coq15] uses several variants of groupings:

- Simple dependent records for integrated groupings.
- *Modules* for stratified groupings, which can be extended by definitions almost arbitrarily using *module functors*, but only linearly in their signature.
- *Structures*
- Type classes [SW11]

HOL Light [Har96] simply uses OCaml files for stratified groupings. Record types are available, but simple and rarely used.

Isabelle [Wen09] has simple records for integrated groupings, and theories for stratified groupings. Both records and record types are exensible.

Locales are structural features which can be used to switch from stratified groupings to an integrated perspective, by generating predicates for the signature of a locale. While locales can be extended by definitions and provable theorems, their signature is fixed. Extending locales is done by redeclaring the corresponding signature in a *sublocale*, which generates the corresponding proof obligations.

Idris [Bra13]

Nuprl [Con+86]

Mizar [TB85]

Maude [Cla+96] Uses *views* from theories to modules to convert from stratified to internalized groupings; although theories are strictly limited to providing (undefined) *signatures*, and hence only serve as interfaces to modules.

OCaml [**ocaml**] Uses modules and signatures for stratified and records for integrated groupings. Modules can be used like values using first-class modules, effectively allowing to treat them like objects in object-oriented languages.

4 Record Types with Defined Fields

We now introduce record types as an additional module of our framework by extending the grammar and the rules. The basic intuition is that $[\![\Gamma]\!]$ and $\langle\Gamma\rangle$ construct record types and terms. We call a context **fully typed** resp. **defined** if all fields have a type resp. a definition. In $[\![\Gamma]\!]$, Γ must be fully typed and may additionally contain defined fields. In $\langle\Gamma\rangle$, Γ must be fully defined; the types are optional and usually omitted in practice.

Because we frequently need fully defined contexts, we introduce a notational convention for them: a context denoted by a *lower* case letters like γ is always fully defined. In contrast, a context denoted by an *upper* case letter like Γ may have any number of types or definitions.

4.1 Records

We extend our grammar as in Figure 6, where the previously existing parts are grayed out.

$$
\begin{array}{lll}
\Gamma ::= \cdot \mid \Gamma, x[:T][:=T] & & \\
T ::= x \mid \texttt{type} \mid \texttt{kind} & & \\
\quad \mid \prod_{x:T'} T \mid \lambda x:T'.T \mid T_1 T_2 & & \\
\quad \mid [\![\Gamma]\!] \mid \langle\Gamma\rangle \mid T.x & & \text{record types, terms, projections}
\end{array}
$$

Fig. 6. Grammar for Records

Remark 2 (Field Names and Substitution in Records). Note that we use the same identifiers for variables in contexts and fields in records. This allows reusing results about contexts when reasoning about and implementing records. In particular, it immediately makes our records dependent, i.e., both in a record type and — maybe surprisingly — in a record term every variable x may occur in subsequent fields. In some sense, this makes x bound in those fields. However, record types are critically different from Σ-types: we must be able to use x in record projections, i.e., x can *not* be subject to α-renaming.

As a consequence, capture-avoiding substitution is not always possible. This is a well-known problem that is usually remedied by allowing every record to declare a name for itself (e.g., the keyword `this` in many object-oriented languages), which is used to disambiguates between record fields and a variable in the surrounding context (or fields in a surrounding record). We gloss over this complication here and simply make substitution a partial function.

Before stating the rules, we introduce a few critical auxiliary definition:

Definition 1 (Substituting in a Record). We extend substitution $t[x/t']$ to records:

– $[\![x_1 : T_1, \ldots, x_n : T_n]\!][y/t]$
$= \begin{cases} [\![x_1 : T_1[y/t], \ldots, x_{i-1} : T_{i-1}[y/t], x_i : T_i, \ldots, x_n : T_n]\!] \text{ if } y = x_i \\ [\![x_1 : (T_1[y/t]), \ldots, x_n : (T_n[y/t])]\!] \text{ else} \end{cases}$
if none of the x_i are free in t. Otherwise the substitution is undefined.
– $\langle x_1 := t_1, \ldots, x_n := t_n \rangle [y/t] = \begin{cases} \langle x_1 := t_1, \ldots, x_n := t_n \rangle \text{ if } y \in \{x_1, \ldots, x_n\} \\ \langle x_1 := (t_1[y/t]), \ldots, x_n := (t_n[y/t]) \rangle \text{ else} \end{cases}$
if none of the x_i are free in t. Otherwise the substitution is undefined.
– $(r.x)[y/t] = (r[y/t]).x$.

Definition 2 (Substituting with a Record). We write $t[r/\Delta]$ for the result of substituting any occurrence of a variable x declared in Δ with $r.x$

In the special case where $r = \langle \delta \rangle$, we simply write $t[\delta]$ for $t[\langle \delta \rangle / \delta]$, i.e., we have $t[x_1 := t_1, \ldots, x_n := t_n] = t[x_n/t_n] \ldots [x_1/t_1]$.

Formation:

$$\frac{\vdash \Gamma, \Delta \ \texttt{ctx} \qquad \Delta \text{ fully typed} \qquad \max \Delta \in \{\texttt{type}, \texttt{kind}\}}{\Gamma \vdash [\![\Delta]\!] \Rightarrow \max \Delta}$$

where $\max \Delta$ is the maximal universe of all undefined fields in Δ

Introduction:

$$\frac{\vdash \Gamma, \Delta \ \texttt{ctx} \quad \delta \text{ fully defined} \quad \Delta \text{ like } \delta \text{ but with all missing types inferred}}{\Gamma \vdash \langle \delta \rangle \Rightarrow [\![\Delta]\!]}$$

Elimination:

$$\frac{\Gamma \vdash r \Rightarrow [\![\Delta_1, x : T[:= t], \Delta_2]\!]}{\Gamma \vdash r.x \Rightarrow T[r/\Delta_1]}$$

Type checking:

$$\frac{\overbrace{\Gamma \vdash r.x \Leftarrow T[r/\Delta]}^{\text{For } x : T[:= t] \in \Delta} \qquad \overbrace{\Gamma \vdash r.x \equiv t[r/\Delta] : T[r/\Delta]}^{\text{additionally for } x : T := t \in \Delta} \qquad \Gamma \vdash r \Rightarrow R}{\Gamma \vdash r \Leftarrow [\![\Delta]\!]}$$

Equality checking (extensionality):

$$\frac{\overbrace{\Gamma \vdash r_1.x \equiv r_2.x : T[r_1/\Delta]}^{\text{For every } (x : T) \in \Delta}}{\Gamma \vdash r_1 \equiv r_2 : [\![\Delta]\!]}$$

Computation:

$$\frac{\delta = \delta_1, x[: T] := t, \delta_2 \quad \Gamma \vdash \langle \delta \rangle \Rightarrow R}{\Gamma \vdash \langle \delta \rangle .x \rightsquigarrow t[\delta_1]} \qquad \frac{\Gamma \vdash r \Rightarrow [\![\Delta_1, x : T := t, \Delta_2]\!]}{\Gamma \vdash r.x \rightsquigarrow t[r/\Delta_1]}$$

Fig. 7. Rules for Records

Our rules for records are given in Figure 7. Their roles are systematically similar to the rules for functions: three inference rules for the three constructors

followed by a type and an equality checking rule for record types and the (in this case: two) computation rules. We remark on a few subtleties below.

The formation rule is partial in the sense that not every context defines a record type or kind. This is because the universe of a record type must be as high as the universe of any undefined field to avoid inconsistencies. For example, $\max(a : \texttt{nat}) = \texttt{type}$, $\max(a : \texttt{type}) = \texttt{kind}$ and $\max(a : \texttt{kind})$ is not defined. If we switched to a countable hierarchy of universes (which we avoid for simplicity), we could turn every context into a record type.

The introduction rule infers the principal type of every record term. Because we allow record types with defined fields, this is the singleton type containing only that record term. This may seem awkward but does not present a problem in practice, where type checking is preferred over type inference anyway.

The elimination rule is straightforward, but it is worth noting that it is entirely parallel to the second computation rule.[4]

The type checking rule has a surprising premise that r must already be well-typed (against some type R). Semantically, this assumption is necessary because we only check the presence of the fields required by $[\![\Delta]\!]$ — without the extra assumption, typing errors in any additional fields that r might have could go undetected. In practice, we implement the rule with an optimization: If r is a variable or a function application, we can efficiently infer some type for it. Otherwise, if $r = \langle\delta\rangle$, some fields of δ have already been checked by the first premise, and we only need to check the remaining fields. The order of premises matters in this case: we want to first use type checking for all fields for which $[\![\Delta]\!]$ provides an expected type before resorting to type inference on the remaining fields.

In the equality checking rule, note that we only have to check equality at undefined fields — the other fields are guaranteed to be equal by the assumption that r_1 and r_2 have type $[\![\Delta]\!]$.

Like the type checking rule, the first computation rule needs the premise that r is well-typed to avoid reducing an ill-typed into a well-typed term. In practice, our framework implements computation with a boolean flag that tracks whether the term to be simplified can be assumed to be well-typed or not; in the former case, this assumption can be skipped.

The second computation rule looks up the definition of a field in the type of the record. Both computation rules can be seen uniformly as definition lookup rules — in the first case the definition is given in the record, in the second case in its type.

Example 1. Figure 8 shows a record type of **Semilattices** (actually, this is a kind because it contains a type field) analogous to the grouping in Figure 1 (using the usual encoding of axioms via judgments-as-types and higher-order abstract syntax for first-order logic).

Then, given a record $r : \texttt{Semilattice}$, we can form the record projection $r.\wedge$, which has type $r.U \to r.U \to r.U$ and $r.\texttt{assoc}$ yields a proof that $r.\wedge$ is

[4] Note that it does not matter how the fields of the record are split into Δ_1 and Δ_2 as long as Δ_1 contains all fields that the declaration of x depends on.

$$\texttt{Semilattice} := \left[\!\!\left[\begin{array}{ll} U & : \texttt{type} \\ \wedge & : U \to U \to U \\ \texttt{assoc} & :\vdash \forall x, y, z : U.\ (x \wedge y) \wedge z \doteq x \wedge (y \wedge z) \\ \texttt{commutative} & : \ldots \\ \texttt{idempotent} & : \ldots \end{array} \right]\!\!\right]$$

Fig. 8. The (Record-)Kind of Semilattices

associative. The intersection on sets forms a semilattice so (assuming we have proofs $\cap - \texttt{assoc}$, $\cap - \texttt{comm}$, $\cap - \texttt{idem}$ with the corresponding types) we can give an instance of that type as

$$\texttt{interSL} : \texttt{Semilattice} := \langle U := \texttt{Set}, \wedge := \cap, \texttt{assoc} := \cap - \texttt{assoc}, \ldots \rangle$$

Theorem 2 (Principal Types). *Our inference rules infer a principal type for each well-typed normal term.*

Proof. Let Γ be a fixed well-typed context. We need to show that for any normal expression t the inferred type is the most specific one, meaning if $\Gamma \vdash t \Rightarrow T$, then for any T' with $\Gamma \vdash t \Leftarrow T'$ we have $\Gamma \vdash T <: T'$.

If the only type checking rule applicable to a term t is an inference rule, then the only way for t to check against a type T' which is not the inferred type T is by first inferring T and then checking $\Gamma \vdash T <: T'$, so in these cases the claim follows by default.

By induction on the grammar:

- $t = x$ and $x : T \in \Gamma$, then $\Gamma \vdash t \Rightarrow T$, which is principal by default.
- $t = \texttt{type}$ then $\Gamma \vdash t \Rightarrow \texttt{kind}$, which is principal by default.
- $t = \texttt{kind}$ is untyped.
- $t = \prod_{x:A} B$ then $\Gamma \vdash t \Rightarrow U$, where $U \in \{\texttt{kind}, \texttt{type}\}$, both of which are principal by default.
- $t = \lambda x : A.t'$ then for some $B : U$ we have $\Gamma, x : A \vdash t' \Rightarrow B$, which is principal by default.
- $t = f\,a$ then $\Gamma \vdash t \Rightarrow B[x/a]$, where $\Gamma \vdash f \Rightarrow \prod_{x:A} B$ $\Gamma \vdash a \Rightarrow A'$ are both principal types and for well-typedness $\Gamma \vdash A' <: A$ needs to hold. Since t is normal, f does not simplify to a lambda-expression and $B[x/a]$ is principal by default.
- $t = [\![\Delta]\!]$ then $\Gamma \vdash t \Rightarrow U$, where $U \in \{\texttt{kind}, \texttt{type}\}$, both of which are unique and hence principal.
- $t = \langle \delta \rangle$ then $\Gamma \vdash r \Rightarrow [\![\Delta]\!]$, wheren Δ contains the exact same variables, but with all types inferred (and by induction hypothesis principal). For t to check against a type, it has to have the form $[\![\Delta']\!]$, hence we need to show that if $\Gamma \vdash t \Leftarrow [\![\Delta']\!]$, then $\Gamma, r : [\![\Delta]\!] \vdash r \Leftarrow [\![\Delta']\!]$.
  Consider the type checking rule for records in Figure 7 and let $x : T[:= d] \in \Delta'$. Since $\Gamma \vdash t \Leftarrow [\![\Delta']\!]$, we have $\Gamma \vdash r.x \Leftarrow T$ (and if x is defined in Δ' also $\Gamma \vdash t.x \equiv d : T$) and since Δ is inferred from t we have $x : T' := d$ in Δ, where by hypothesis T' is the principal type of d and hence $\Gamma \vdash T' <: T$.

As a result, $\Gamma, r : [\![\Delta]\!] \vdash r.x \Rightarrow T'$, therefore $\Gamma, r : [\![\Delta]\!] \vdash r.x \Leftarrow T$ and $\Gamma, r : [\![\Delta]\!] \vdash r.x \equiv d : T$ and hence $\Gamma, r : [\![\Delta]\!] \vdash r \Leftarrow [\![\Delta']\!]$, which makes $[\![\Delta]\!]$ the principal type of t.

$t = r.x$ Since t is normal, we have $\Gamma \vdash r.x \Rightarrow T[r/\Delta_1]$ for some $\Gamma \vdash r \Rightarrow [\![\Delta_1, x : T, \Delta_2]\!]$. Since the latter is by hypothesis the principal type of r, for t to typecheck against some T' it needs to be the case that r type checks against some $R = [\![\Delta_1', x : T', \Delta_2']\!]$ and $\Gamma \vdash T <: T'$ holds by the principal type of r.

In analogy to function types, we can derive the subtyping properties of record types. We introduce context subsumption and then combine horizontal and vertical subtyping in a single statement.

Definition 3 (Context Subsumption). For two fully typed contexts Δ_i we write $\Gamma \vdash \Delta_1 \hookrightarrow \Delta_2$ iff for every declaration $x : T[:= t]$ in Δ_1 there is a declaration $x : T'[:= t']$ in Δ_2 such that

- $\Gamma \vdash T' <: T$ and
- if t is present, then so is t' and $\Gamma \vdash t \equiv t' : T$

Intuitively, $\Delta_1 \hookrightarrow \Delta_2$ means that everything of Δ_1 is also in Δ_2. That yields:

Theorem 3 (Record Subtyping). *The following rule is derivable:*

$$\frac{\Gamma \vdash \Delta_1 \hookrightarrow \Delta_2}{\Gamma \vdash [\![\Delta_2]\!] <: [\![\Delta_1]\!]}$$

Proof. Assume $\Gamma \vdash \Delta_1 \hookrightarrow \Delta_2$. We need to show $\Gamma, r : [\![\Delta_2]\!] \vdash r \Leftarrow [\![\Delta_1]\!]$. By the type checking rule in Figure 7, for any $x : T[:= t] \in \Delta_1$, we need to show that $\Gamma, r : [\![\Delta_2]\!] \vdash r.x \Leftarrow T$ (and if applicable $\Gamma, r : [\![\Delta_2]\!] \vdash r.x \equiv t : T$).

By definition of $\Delta_1 \hookrightarrow \Delta_2$, since $x : T[:= t] \in \Delta_1$, we have $x : T'[:= t] \in \Delta_2$ and $\Gamma \vdash T' <: T$, and if x is defined in Δ_1 the required equality holds as well, so the type checking rule proves $\Gamma, r : [\![\Delta_2]\!] \vdash r \Leftarrow [\![\Delta_1]\!]$ and the claim follows.

4.2 Merging Records

We introduce an advanced operation on records, which proves critical for both convenience and performance: Theories can easily become very large containing hundreds or even thousands of declarations. If we want to treat theories as record types, we need to be able to build big records from smaller ones without exploding them into long lists. Therefore, we introduce an explicit merge operator + on both record types and terms.

In the grammar, this is a single production for terms:

$$T ::= T + T$$

The intended meaning of + is given by the following definition:

Definition 4 (Merging Contexts). Given a context Δ and a (not necessarily well-typed) context E, we define a partial function $\Delta \oplus E$ as follows:

- $\cdot \oplus E = E$
- If $\Delta = d, \Delta_0$ where d is a single declaration for a variable x:
  - if x is not declared in E: $(d, \Delta_0) \oplus E = d, (\Delta_0 \oplus E)$
  - if $E = E_0, e, E_1$ where e is a single declaration for a variable x:
    - if a variable in E_0 is also declared in Δ_0: $\Delta \oplus E$ is undefined,
    - if d and e have unequal types or unequal definitions: $\Delta \oplus E$ is undefined[5],
    - otherwise, $(d, \Delta_0) \oplus (E_0, e, E_1) = E_0, m, (\Delta_0, E_1)$ where m arises by merging d and e.

Note that $\oplus$ is an asymmetric operator: While Δ must be well-typed (relative to some ambient context), E may refer to the names of Δ and is therefore not necessarily well-typed on its own.

We do not define the semantics of + via inference and checking rules. Instead, we give equality rules that directly expand + into $\oplus$ when possible:

$$\frac{\vdash \Gamma, (\Delta_1 \oplus \Delta_2)\ \texttt{ctx}}{\Gamma \vdash [\![\Delta_1]\!] + [\![\Delta_2]\!] \rightsquigarrow [\![\Delta_1 \oplus \Delta_2]\!]} \qquad \frac{\vdash \Gamma, (\delta_1 \oplus \delta_2)\ \texttt{ctx}}{\Gamma \vdash \langle \delta_1 \rangle + \langle \delta_2 \rangle \rightsquigarrow \langle \delta_1 \oplus \delta_2 \rangle}$$

$$\frac{\vdash \Gamma, (\Delta \oplus \delta)\ \texttt{ctx}}{\Gamma \vdash [\![\Delta]\!] + \langle \delta \rangle \rightsquigarrow \langle \Delta \oplus \delta \rangle}$$

In implementations some straightforward optimizations are needed to verify the premises of these rules efficiently; we omit that here for simplicity. For example, merges of well-typed records with disjoint field names are always well-typed, but e.g., $[\![x : nat]\!] + [\![x : bool]\!]$ is not well-typed even though both arguments are.

In practice, we want to avoid using the computation rules for + whenever possible. Therefore, we prove admissible rules (i.e., rules that can be added without changing the set of derivable judgments) that we use preferentially:

Theorem 4. *If R_1, R_2, and $R_1 + R_2$ are well-typed record types, then $R_1 + R_2$ is the greatest lower bound with respect to subtyping of R_1 and R_2. In particular, $\Gamma \vdash r \Leftarrow R_1 + R_2$ iff $\Gamma \vdash r \Leftarrow R_1$ and $\Gamma \vdash r \Leftarrow R_2$*

If $\Gamma \vdash r_i \Leftarrow R_i$ and $r_1 + r_2$ is well-typed, then $\Gamma \vdash r_1 + r_2 \Leftarrow R_1 + R_2$

Proof.
- If $R_1 = [\![\Delta_1]\!]$ and $R_2 = [\![\Delta_2]\!]$ are well-typed record types, then $R_1 + R_2 \rightsquigarrow [\![\Delta_1 \oplus \Delta_2]\!]$. By definition of $\oplus$, any $r : R_1 + R_2$ hence has all fields defined that are required by both $[\![\Delta_1]\!]$ and $[\![\Delta_2]\!]$ and the other way around.
- By the rules for +, for $r_1 = \langle \delta_1 \rangle$ and $\langle \delta_2 \rangle$ we get $r_1 + r_2 = \langle \delta_1 \oplus \delta_2 \rangle$ and if $R_1 = [\![\Delta_1]\!]$ and $R_2 = [\![\Delta_2]\!]$, then (since $\Gamma \vdash r_i \Leftarrow R_i$) we have $\Delta_1 \hookrightarrow \delta_1$ and $\Delta_2 \hookrightarrow \delta_2$.
  As can easily be verified, it follows that $\Delta_1 \oplus \Delta_2 \hookrightarrow \delta_1 \oplus \delta_2$, and hence $\langle \delta_1 \oplus \delta_2 \rangle \Rightarrow [\![\delta_1 \oplus \delta_2]\!] <: [\![\Delta_1 \oplus \Delta_2]\!] = R_1 + R_2$.

Inspecting the type checking rule in Figure 7, we see that a record r of type $[\![\Delta]\!]$ must repeat all defined fields of Δ. This makes sense conceptually but would

[5] It is possible and important in practice to also define $\Delta \oplus E$ when the types/definitions in d and e are provably equal. We omit that here for simplicity.

be a major inconvenience in practice. The merging operator solves this problem elegantly as we see in the following example:

Example 2. Continuing our running example, we can now define a type of semilattices *with order* (and all associated axioms) as in Figure 9.

$$\texttt{SemilatticeOrder} := \texttt{Semilattice} + \left[\!\!\left[\begin{array}{ll} \leq & : U \to U \to U := \lambda x, y : U.\ x \doteq x \wedge y \\ \texttt{refl} & :\vdash \forall a : U.\ a \leq a := (\text{proof}) \\ \ldots & \end{array} \right]\!\!\right]$$
$$\texttt{interSLO} := \texttt{SemilatticeOrder} + \texttt{interSL}$$

Fig. 9. Running Example

Now the explicit merging in the type `SemilatticeOrder` allows the projection $\texttt{interSLO}.\leq$, which is equal to $\lambda x, y : (\texttt{interSLO}.U)\ .\ (x \doteq x(\texttt{interSLO}.\wedge)y)$ and $\texttt{interSLO.refl}$ yields a proof that this order is reflexive – without needing to define the order or prove the axiom anew for the specific instance `interSL`.

5 Internalizing Theories

5.1 Preliminaries: Theories

$$\begin{array}{lll} \Theta ::= \cdot \mid \Theta,\ X = \{\Gamma\} \mid \Theta,\ X : X_1 \to X_2 = \{\Gamma\} & & \text{theory level} \\ \Gamma ::= \cdot \mid \Gamma, x[: T][:= T] \mid \Gamma,\ \texttt{include}\, X & & \text{includes} \\ T ::= x \mid \texttt{type} \mid \texttt{kind} & & \\ \quad \mid \prod_{x:T'} T \mid \lambda x : T'.T \mid T_1 T_2 & & \\ \quad \mid [\![\Gamma]\!] \mid \langle\Gamma\rangle \mid T.x \mid T_1 + T_2 & & \end{array}$$

Fig. 10. A Simple Stratified Language

We introduce a minimal definition of stratified theories and theory morphisms, which can be seen as a very simple fragment of the MMT language [RK13]. The **grammar** is given in Figure 10, again graying out the previously introduced parts.

Each of the two levels has its own context: Firstly, the **theory level** context Θ introduces names X, which can be either theories $X = \{\Gamma\}$ or morphisms $X : P \to Q = \{\Gamma\}$, where P and Q are the names of previously defined theories. Secondly, the **expression level** context Γ is as before but may additionally contain includes $\texttt{include}\, X$ of other theories resp. morphisms X. We call a context **flat** if it does not contain includes.

All **judgments** are as before except that they acquire a second context, e.g., the typing judgment now becomes $\Theta; \Gamma \vdash t \Leftarrow T$. With this modification, all rules for function and record types remain unchanged. However, we add the restriction that Γ in $[\![\Gamma]\!]$ and $\langle\Gamma\rangle$ must be flat.

We omit the **rules** for theories and morphisms for brevity and only sketch their intuitions. We think of theories as named contexts and of morphisms as named substitutions between contexts. Includes allow forming both modularly by copying over the declarations of a previously named object. While theories may contain arbitrary declarations, morphisms are restricted: Let Θ contain $P = \{\Gamma\}$ and $Q = \{\Delta\}$. Then a morphism $V : P \to Q = \{\delta\}$ is well-typed if δ is fully defined (akin to record terms) and contains for each declaration $x : T$ of P a declaration $x = t$ where t may refer to all names declared in Q. V induces a homomorphic extension $\overline{V}$ that maps P-expressions to Q-expressions. The key property of morphisms is that, if V is well-typed, then $\Theta; P \vdash t \Leftarrow T$ implies $\Theta; Q \vdash \overline{V}(t) \Leftarrow \overline{V}(T)$ and accordingly for equality checking and subtyping. Thus, theory morphisms preserve judgments and (via propositions-as-types representations) truth. Moreover, it is straightforward to extend the above with identity and composition so that theories and morphisms form a category. We refer to [Rab17b] for details.

5.2 Internalization

We can now add the internalization operator, for which everything so far was preparation. We add one production to the **grammar**:

$$T ::= \texttt{Mod}\,(X)$$

The intended meaning of $\texttt{Mod}\,(X)$ is that it turns a theory X into a record type and a morphism $X : P \to Q$ into a function $\texttt{Mod}\,(Q) \to \texttt{Mod}\,(P)$. For simplicity, we only state the rules for the case where all include declarations are at the beginning of theory/morphism:

$$\frac{P = \{\texttt{include}\,P_1, \ldots, \texttt{include}\,P_n, \Delta\} \text{ in } \Theta \quad \Delta \text{ flat} \quad \max P \text{ defined}}{\Theta; \Gamma \vdash \texttt{Mod}\,(P) \rightsquigarrow \texttt{Mod}\,(P_1) + \ldots + \texttt{Mod}\,(P_n) + [\![\Delta]\!]}$$

$$\frac{V : P \to Q = \{\texttt{include}\,V_1, \ldots, \texttt{include}\,V_n, \delta\} \text{ in } \Theta \quad \delta \text{ flat}}{\Theta; \Gamma \vdash \texttt{Mod}\,(V) \rightsquigarrow \lambda r : \texttt{Mod}\,(Q)\,.\,\texttt{Mod}\,(P) + (\texttt{Mod}\,(V_1)\,r) + \ldots + (\texttt{Mod}\,(V_n)\,r) + \langle \delta[r] \rangle}$$

where we use the following abbreviations:

- In the rule for theories, $\max P$ is the biggest universe occurring in any declaration transitively included into P, i.e., $\max P = \max\{\max P_1, \ldots, \max P_n, \max \Delta\}$ (undefined if any argument is).
- In the rule for morphisms, $\delta[r]$ is the result of substituting in δ every reference to a declaration of x in Q with $r.x$.

In the rule for morphisms, the occurrence of $\texttt{Mod}\,(P)$ may appear redundant; but it is critical to (i) make sure all defined declarations of P are part of the record and (ii) provide the expected types for checking the declarations in δ.

Example 3. Consider the theories in Figure 11. Applying $\texttt{Mod}\,(\cdot)$ to these theories yields exactly the record types of the same name introduced in Section 4 (Figures 8 and 9), i.e., we have $\texttt{interSL} \Leftarrow \texttt{Mod}\,(\texttt{Semilattice})$ and $\texttt{interSLO} \Leftarrow \texttt{Mod}\,(\texttt{SemilatticeOrder})$. In particularly, $\texttt{Mod}$ preserves the modular structure of the theory.

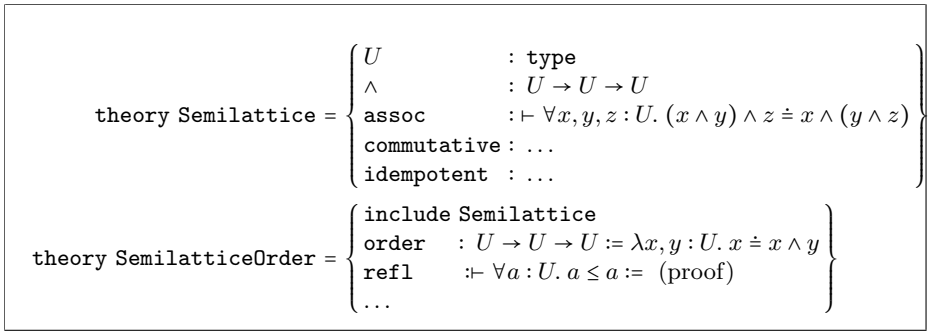

Fig. 11. A Theory of Semilattices

The basic properties of $\texttt{Mod}\,(X)$ are collected in the following theorem:

Theorem 5 (Functoriality). $\texttt{Mod}\,(\cdot)$ *is a monotonic contravariant functor from the category of theories and morphisms ordered by inclusion to the category of types (of any universe) and functions ordered by subtyping. In particular,*

- *if* P *is a theory in* Θ *and* $\max P \in \{\texttt{type}, \texttt{kind}\}$*, then* $\Theta; \Gamma \vdash \texttt{Mod}\,(P) \Leftarrow \max P$
- *if* $V : P \to Q$ *is a theory morphism in* $\Theta; \Gamma \vdash \texttt{Mod}\,(V) \Leftarrow \texttt{Mod}\,(Q) \to \texttt{Mod}\,(P)$
- *if* P *is transitively included into* Q*, then* $\Theta; \Gamma \vdash \texttt{Mod}\,(Q) <: \texttt{Mod}\,(P)$

Proof.
- Follows immediately by the computation rule for $\texttt{Mod}\,(P)$.
- Follows immediately by the computation rule for $\texttt{Mod}\,(V)$ and the type checking rule for λ.
- Follows immediately by the computation rule for $\texttt{Mod}\,(P)$ and Theorem 4.

An immediate advantage of $\texttt{Mod}\,(\cdot)$ is that we can now use the expression level to define expression-like theory level operations. As an example, we consider the **intersection** $P \cap P'$ of two theories, i.e., the theory that includes all theories included by both P and P'. Instead of defining it at the theory level, which would begin a slippery slope of adding more and more theory level operations, we can simply build it at the expression level:

$$P \cap P' := \texttt{Mod}\,(Q_1) + \ldots + \texttt{Mod}\,(Q_n)$$

where the Q_i are all theories included into both P and P'.[6]

Note that the computation rules for $\texttt{Mod}$ are efficient in the sense that the structure of the theory level is preserved. In particular, we do not flatten theories and morphisms into flat contexts, which would be a huge blow-up for big theories.[7]

[6] Note that because $P \cap P'$ depends on the syntactic structure of P and P', it only approximates the least upper bound of $\texttt{Mod}\,(P)$ and $\texttt{Mod}\,(P')$ and is not stable under, e.g., flattening of P and P'. But it can still be very useful in certain situations.

[7] The computation of $\max P$ may look like it requires flattening. But it is easy to compute and cache its value for every named theory.

However, efficiently *creating* the internalization is not enough. $\texttt{Mod}\,(X)$ is defined via $+$, which is itself only an abbreviation whose expansion amounts to flattening. Therefore, we establish admissible rules that allow *working with* internalizations efficiently, i.e., without computing the expansion of $+$:

Theorem 6. *Fix well-typed Θ, Γ and $P = \{\texttt{include}\,P_1, \dots, \texttt{include}\,P_n, \Delta\}$ in Θ. Then the following rules are admissible:*

$$\frac{\overbrace{\Theta;\Gamma \vdash r \Leftarrow \texttt{Mod}\,(P_i)}^{1\le i\le n} \quad \overbrace{\Theta;\Gamma \vdash r.x \Leftarrow T[r/P]}^{x:T\in\Delta} \quad \overbrace{\Theta;\Gamma \vdash r.x \equiv t[r/P] : T[r/P]}^{x:T:=t\in\Delta} \quad \Gamma \vdash r \Rightarrow R}{\Theta;\Gamma \vdash r \Leftarrow \texttt{Mod}\,(P)}$$

$$\frac{\overbrace{\Theta;\Gamma \vdash r_i \Leftarrow \texttt{Mod}\,(P_i)}^{1\le i\le n} \quad \overbrace{\Theta;\Gamma \vdash r_i \equiv r_j : P_i \cap P_j}^{1\le i,j\le n} \quad \Theta;\Gamma \vdash \langle\delta\rangle\,[r/P] \Leftarrow [\![\Delta]\!] \quad \Gamma \vdash r \Rightarrow R}{\Theta;\Gamma \vdash \underbrace{\texttt{Mod}\,(P) + r_1 + \ldots + r_n + \langle\delta\rangle}_{=:r} \Rightarrow \texttt{Mod}\,(P)}$$

where $[r/P]$ abbreviates the substitution that replaces every x declared in a theory transitively-included into P with $r.x$.[8]

The first rule in Theorem 6 uses the modular structure of P to check r at type $\texttt{Mod}\,(P)$. If r is of the form $\langle\delta\rangle$, this is no faster than flattening $\texttt{Mod}\,(P)$ all the way. But in the typical case where r is also formed modularly using a similar structure as P, this can be much faster. The second rule performs the corresponding type inference for an element of $\texttt{Mod}\,(P)$ that is formed following the modular structure of P. In both cases, the last premise is again only needed to make sure that r does not contain ill-typed fields not required by $\texttt{Mod}\,(P)$. Also note that if we think of $\texttt{Mod}\,(P)$ as a colimit and of elements of $\texttt{Mod}\,(P)$ as morphisms out of P, then the second rule corresponds to the construction of the universal morphisms out of the colimit.

Example 4. We continue Ex. 3 and assume we have already checked $\texttt{interSL} \Leftarrow \texttt{Mod}\,(\texttt{Semilattice})$ (*).

We want to check $\texttt{interSL} + \langle\delta\rangle \Leftarrow \texttt{Mod}\,(\texttt{SemilatticeOrder})$. Applying the first rule of Thm. 6 reduces this to multiple premises, the first one of which is (*) and can thus be discharged without inspecting $\texttt{interSL}$.

Ex. 4 is still somewhat artificial because the involved theories are so small. But the effect pays off enormously on larger theories.

Additionally, we can explicitly allow views *within* theories into the current theory. Specifically, given a theory T, we allow a view

$$T = \{\ldots, V : T' \to \cdot = \{\ldots\}, \ldots \quad\}$$

[8] In practice, these substitutions are easy to implement without flattening r because we can cache for every theory which theories it includes and which names it declares.

the codomain of which is the containing theory T (up to the point where V is declared). [9] This view induces a view $T/V : T' \to T$ in the top-level context Θ, but importantly, within T (and its corresponding inner context Γ) every variable in V is defined via a valid term in Γ. Correspondingly, $\texttt{Mod}\,(V)$ is – in context Γ – a constant function $\texttt{Mod}\,(T) \to \texttt{Mod}\,(T')$ which we can consider as an element of $\texttt{Mod}\,(T')$ directly.

This allows for conveniently building instances of $\texttt{Mod}\,(\cdot)$ and all checking for well-formedness is reduced to structurally checking the view to be well-formed, effectively carrying over all efficiency advantages of structure checking and modular development of theories/views:

Theorem 7. *Let Θ, Γ be well-formed and $T/V : T' \to T = \{\ldots\}$ in Θ, where Γ is the current context within theory T containing $V : T' \to \cdot = \{\ldots\}$. The following rule is admissible:*

$$\frac{}{\Theta; \Gamma \vdash \texttt{Mod}\,(V) \Leftarrow \texttt{Mod}\,(T')}$$

Proof. We consider $\texttt{Mod}\,(V)$ an abbreviation for $\texttt{Mod}\,(T/V)\,(\langle\cdot\rangle)$. Since all definitions in $\texttt{Mod}\,(T/V)$ are well-typed terms in context Γ, the record $\langle\cdot\rangle$ does not actually occur anywhere in the simplified application $\texttt{Mod}\,(T/V)\,(\langle\cdot\rangle)$, which makes this expression well-typed.

6 Implementation and Case Study

We have implemented a variant of the record types and the $\texttt{Mod}\,(\cdot)$-operator described here in the MMT-system (as part of [LFX]). They are used extensively in the *Math-in-the-Middle* archive (MitM), which forms an integral part in the OpenDreamKit [Deh+16] and MaMoRed [Koh+17] projects. In particular the formalizations of algebra and topology are systematically built on top of the concepts presented in this paper.

The archive sources can be found at [Mit], and its contents can be inspected and browsed online at `https://mmt.mathhub.info` under MitM/smglom. Note that the $\texttt{Mod}\,(\cdot)$ operator is called `ModelsOf` here.

For a particularly interesting example that occurs in MitM, consider the theories for modules and vector spaces (over some ring/field) given in Figure 12, which elegantly follow informal mathematical practice. Going beyond the syntax introduced so far, these use *parametric* theories. Our implementation extends `Mod` to parametric theories as well, namely in such a way that $\texttt{Mod}\,(\texttt{Module}) : \prod_{R:\texttt{Mod}(\texttt{Ring})} \texttt{Mod}\,(\texttt{Module}(R))$ and correspondingly for fields. Thus, we obtain

$$\texttt{Mod}\,(\texttt{VSpace}) = \lambda F : \texttt{Mod}\,(\texttt{Field})\,.((\texttt{Mod}\,(\texttt{Module})\ F) + \ldots)$$

and, e.g., $\texttt{Mod}\,(\texttt{VSpace})\ \mathbb{R} <: \texttt{Mod}\,(\texttt{Module})\ \mathbb{R}$. Because of type-level parameters, this requires some kind of parametric polymorphism in the type system. For our approach, the shallow polymorphism module that is available in MMT is sufficient.

[9] We'll omit the adapted grammar for now.

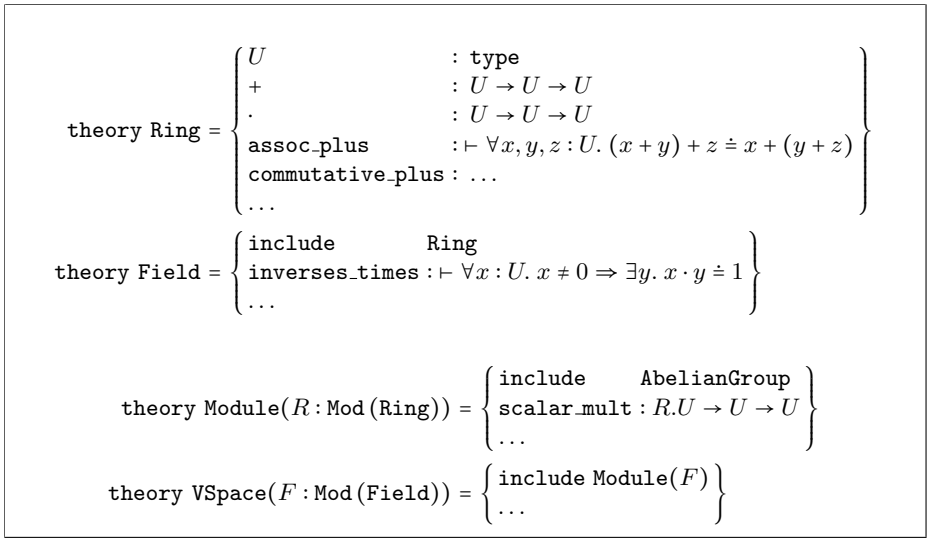

Fig. 12. Theories for R-Modules and Vector Spaces

7 Conclusion

We have presented a formal system that allows to systematically combine the advantages of stratified and integrated grouping mechanisms found in type theories, logics, and specification/programming languages. Concretely, our system allows internalizing theories into record types in a way that preserves their defined fields and modular structure.

Our MitM case study shows that theory internalization is an important feature of any foundation; especially if it interfaces to differing mathematical software systems. Our experiments have also shown that (predicate) subtyping makes internalization even stronger in practice. But type-inference in the combined system induces non-trivial trade-offs; which we leave to future work.

Acknowledgements The work reported here has been kicked off by discussions with Jacques Carette and William Farmer who have experimented with theory internalizations into record types in the scope of their MathScheme system. We acknowledge financial support from the OpenDreamKit Horizon 2020 European Research Infrastructures project (#676541).

References

[Bra13] E. Brady. "Idris, a general-purpose dependently typed programming language: Design and implementation". In: *Journal of Functional Programming* 23.5 (2013), pp. 552–593.

[Cla+96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. "Principles of Maude". In: *Proceedings of the First International Workshop on Rewriting Logic*. Ed. by J. Meseguer. Vol. 4. 1996, pp. 65–89.

[Con+86] R. Constable et al. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986.

[Deh+16] P.-O. Dehaye et al. "Interoperability in the OpenDreamKit Project: The Math-in-the-Middle Approach". In: *Intelligent Computer Mathematics 2016*. Ed. by M. Kohlhase, M. Johansson, B. Miller, L. de Moura, and F. Tompa. LNAI 9791. Springer, 2016. URL: https://github.com/OpenDreamKit/OpenDreamKit/blob/master/WP6/CICM2016/published.pdf.

[FGT92] W. Farmer, J. Guttman, and F. Thayer. "Little Theories". In: *Conference on Automated Deduction*. Ed. by D. Kapur. 1992, pp. 467–581.

[Gog+93] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J. Jouannaud. "Introducing OBJ". In: *Applications of Algebraic Specification using OBJ*. Ed. by J. Goguen, D. Coleman, and R. Gallimore. Cambridge, 1993.

[Har+12] T. Hardin et al. *The FoCaLiZe Essential*. http://focalize.inria.fr/. 2012.

[Har96] J. Harrison. "HOL Light: A Tutorial Introduction". In: *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*. Springer, 1996, pp. 265–269.

[Koh+17] M. Kohlhase, T. Koprucki, D. Müller, and K. Tabelow. "Mathematical models as research data via flexiformal theory graphs". In: *Intelligent Computer Mathematics (CICM) 2017*. Ed. by H. Geuvers, M. England, O. Hasan, F. Rabe, and O. Teschke. LNAI 10383. Springer, 2017. DOI: 10.1007/978-3-319-62075-6.

[LFX] *MathHub MMT/LFX Git Repository*. URL: http://gl.mathhub.info/MMT/LFX (visited on 05/15/2015).

[Luo09] Z. Luo. "Manifest Fields and Module Mechanisms in Intensional Type Theory". In: *Types for Proofs and Programs*. Ed. by S. Berardi, F. Damiani, and U. de'Liguoro. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 237–255.

[Mat] *Mathematical Components*. URL: http://www.msr-inria.fr/projects/mathematical-components-2/.

[Mit] *MitM/smglom*. URL: https://gl.mathhub.info/MitM/smglom (visited on 02/01/2018).

[MRK18] D. Müller, F. Rabe, and M. Kohlhase. "Theories as Types". In: ed. by D. Galmiche, S. Schulz, and R. Sebastiani. Springer Verlag, 2018. URL: http://kwarc.info/kohlhase/papers/ijcar18-records.pdf.

[Nor05] U. Norell. *The Agda WiKi*. http://wiki.portal.chalmers.se/agda. 2005.

[ORS92] S. Owre, J. Rushby, and N. Shankar. "PVS: A Prototype Verification System". In: *11th International Conference on Automated Deduction (CADE)*. Ed. by D. Kapur. Springer, 1992, pp. 748–752.

[Rab17a] F. Rabe. "A Modular Type Reconstruction Algorithm". In: *ACM Transactions on Computational Logic* (2017). accepted pending minor revision; see https://kwarc.info/people/frabe/Research/rabe_recon_17.pdf.

[Rab17b] F. Rabe. "How to Identify, Translate, and Combine Logics?" In: *Journal of Logic and Computation* 27.6 (2017), pp. 1753–1798.

[RK13] F. Rabe and M. Kohlhase. "A Scalable Module System". In: *Information and Computation* 230.1 (2013), pp. 1–54.

[SW11] B. Spitters and E. van der Weegen. "Type Classes for Mathematics in Type Theory". In: *CoRR* abs/1102.1323 (2011). arXiv: 1102.1323. URL: http://arxiv.org/abs/1102.1323.

[TB85] A. Trybulec and H. Blair. "Computer Assisted Reasoning with MIZAR". In: *Proceedings of the 9th International Joint Conference on Artificial Intelligence*. Ed. by A. Joshi. Morgan Kaufmann, 1985, pp. 26–28.

[Wen09] M. Wenzel. *The Isabelle/Isar Reference Manual*. http://isabelle.in.tum.de/documentation.html, Dec 3, 2009. 2009.

[Coq15] Coq Development Team. *The Coq Proof Assistant: Reference Manual*. Tech. rep. INRIA, 2015.