Intensional Manifest Fields in Module Types[☆]

Zhaohui Luo

Department of Computer Science
Royal Holloway, University of London
Egham, Surrey TW20 0EX, U.K.

Abstract

A manifest field in a type of modules is a field whose expected data is not only of a certain type but the same as a specific object of that type. All of the previous approaches to manifest fields in type theory are based on some extensional notions of computational equality. In this paper, we show that this is unnecessary: manifest fields are expressible in intensional type theories without extensional equality rules. These *intensional manifest fields* are made available with the help of coercive subtyping. It is shown that, for both Σ-types and dependent record types, the `with`-clause for expressing manifest fields can be introduced by means of the intensional manifest fields. This provides an internal mechanism in intensional type theories to express definitional entries in module types, which has useful applications including, for example, the representation of higher-order modules with ML-style sharing.

Keywords: Manifest field, Intensional type theory, Module, Σ-type, Dependent record type

1. Introduction

Modularity is important in programming and proving in the large and the module mechanism is essential in both programming and proof languages. In type theory, a type of modules may be expressed as a Σ-type of unlabelled tuples or a dependent record type of labelled tuples. A field in such a type is usually *abstract* (of the form '$v : A$') in the sense that the data expected in

[☆]This work was partially supported by the research grant F/07-537/AA of the Leverhulme Trust in U.K. and the TYPES grant IST-510996 of EU.

that field can be any object of a given type. In contrast, a field is *manifest* (of the form '$v = a : A$') if the data expected in that field is not only of a given type but the 'same' as some specific object of that type. If available, manifest fields provide a powerful representation mechanism which intuitively allows an internal expression of definitional entries at the level of types: one can use it to express the typing constraint that a field expects a particular object of a type rather than anyone of that type. For instance, one can use Σ-types or dependent record types with manifest fields to express the powerful module mechanism with the so-called ML-style sharing (or sharing by equations) [35, 22].

For a manifest field $v = a : A$, the 'sameness' of the expected data as a may be interpreted as that it is computationally equal to a, as is done in most, if not all, of the previous studies on manifest fields in type theory (see, for example, [18, 40, 12]). This gives rise to an extensional notion of equality: the data in a manifest field $v = a : A$, even when it is a variable, is expected to be computationally equal to a. Such manifest fields may be called *extensional manifest fields*. In type theory, extensional manifest fields may also be obtained by means of other extensional constructs such as the singleton type [3, 19, 44] and the (strong) extensional equality [36, 8]. It is known, however, such an extensional notion of equality is meta-theoretically difficult (in the cases of the extensional manifest fields and the singleton types) or even leads to an outright undecidability (in the case of the extensional equality).

As shown in this paper, the 'sameness' for a manifest field does not have to be interpreted by means of an extensional equality. With the help of coercive subtyping [27], manifest fields are expressible in *intensional* type theories such as Martin-Löf's intensional type theory [39] and UTT [25]. The idea is very simple: a manifest field $v = a : A$ is simply expressed as the shorthand of an ordinary (abstract) field $v : \mathbf{1}(A, a)$, where $\mathbf{1}(A, a)$ is the inductive unit type parameterised by A and a. Then, with a coercion that maps the objects of $\mathbf{1}(A, a)$ to a, v stands for a in any context that requires an object of type A. This achieves exactly what we want with a manifest field. Such a manifest field may be called an *intensional manifest field* and, to distinguish it from an extensional manifest field, we use the notation $v \sim a : A$ to stand for $v : \mathbf{1}(A, a)$.

Intensional manifest fields (IMFs for short) can be introduced for both Σ-types and dependent record types. This paper develops IMFs for both kinds of the module types. It also shows how the so-called `with`-clause [40],

that intuitively expresses that a field is manifest rather than abstract, can be defined by means of IMFs and used as expected in the presence of the component-wise coercions that propagate subtyping relations through the module types. In order to study IMFs for dependent record types, we consider an improved formulation based on that of record types [40] rather than record kinds [5, 12].[1] The formulation of dependent record types has its own merits but can only be discussed briefly in this paper (see [29] for further information).

Intensional manifest fields can be used to express definitional entries and provide a higher-order module mechanism with ML-style sharing[2]. Using the record macro in Coq [10], we give examples to show how ML-style sharing can be captured with IMFs in a proof assistant that implements an intensional type theory. Since many of the current proof assistants support the use of coercions, the module mechanism supported by IMFs can be used for modular development of proofs and dependently-typed programs.

The following subsection describes briefly the background and notational conventions. In Section 2, after introducing manifest fields, we show how IMFs can be introduced for Σ-types in an intensional type theory. In Section 3, we formulate dependent record types, introduce the IMFs in record types, and illustrate their use in expressing the module mechanism with ML-style sharing. The related and future work is discussed in the conclusion.

1.1. Background and Notational Conventions: LF and Coercive Subtyping

In this subsection, we give a brief description of the logical framework LF and the framework of coercive subtyping, partly to introduce the background and partly to fix notational conventions.

The logical framework LF. LF [25] is the typed version of Martin-Löf's logical framework [39]. It is a dependent type system for specifying type theories such as Martin-Löf's intensional type theory [39], the Calculus of Constructions (CC) [11] and the Unifying Theory of dependent Types (UTT) [25]. The types in LF are called *kinds*, including:

[1] *Types* in the terminology of Martin-Löf's type theory are what we call *kinds* in this paper. Therefore, the so-called record types in [5] and [12] are really record kinds.

[2] Historically, expressing ML-style sharing was a main motivation behind the studies of manifest fields [18, 22, 40]. In fact, many people have long believed that, to express ML-style sharing in type theory, it is essential to have some construct with an extensional notion of equality. As shown in this paper, this is actually unnecessary.

- $Type$ – the kind representing the universe of types (A is a type if $A : Type$);
- $El(A)$ – the kind of objects of type A (we often omit El); and
- $(x{:}K)K'$ (or simply $(K)K'$ when $x \notin FV(K')$) – the kind of dependent functional operations such as the abstraction $[x{:}K]k'$.

The rules of LF can be found in Appendix A.

Notations We shall adopt the following conventions.

- Substitution: We sometimes use $M\{x\}$ to indicate that x may occur free in M and subsequently write $M\{a\}$ for the substitution $[a/x]M$.
- Functional composition: For $f : (K_1)K_2$ and $g : (K_2)K_3$, $g \circ f = [x{:}K_1]g(f(x)) : (K_1)K_3$, where x does not occur free in f or g.

When a type theory is specified in LF, its types are declared, together with their introduction/elimination operators and the associated computation rules. Examples of types include

- inductive types such as Nat of natural numbers,
- inductive families of types such as $Vect(n)$ of vectors of length n, and
- families of inductive types such as Π-types and Σ-types.

A Π-type $\Pi(A, B)$ is the type of functions $\lambda(x{:}A)b$ and a Σ-type $\Sigma(A, B)$ of dependent pairs (a, b). (We use $A \to B$ and $A \times B$ for the non-dependent Π-type and Σ-type, respectively. Also, see Appendix B for a further explanation for the notation of untyped pairs.) In a non-LF notation, $\Sigma(A, B)$, for example, will be written as $\Sigma x{:}A.B(x)$.

A nested Σ-type can be seen as a type of tuples/modules. We shall adopt the following notational convention.

Notation For $n \geq 1$, we shall use

$$\sum[x_1 : A_1,\ x_2 : A_2,\ ...,\ x_n : A_n]$$

to stand for the following Σ-type:

$$\Sigma x_1 : A_1 \Sigma x_2 : A_2\ ...\ \Sigma x_{n-1} : A_{n-1}.\ A_n.$$

When $n = 1$, $\sum[x{:}A] =_{\rm df} A$. Also, we shall use the tuple

$$(a_1,\ a_2,\ ...,\ a_n)$$

to stand for the nested pair

$$(a_1,\ (a_2,\ ...,\ (a_{n-1}, a_n)...)).$$

Furthermore, for any a of type $\sum[x_1 : A_1,\ x_2 : A_2,\ ...,\ x_n : A_n]$,

- $a.i =_{\rm df} \pi_1(\pi_2(...\pi_2(a)...)$, where π_2 occurs $i-1$ times $(1 \leq i < n)$, and
- $a.n =_{\rm df} \pi_2(...\pi_2(a)...)$, where π_2 occurs $n-1$ times.

For instance, when $n = 3$, $a.2 \equiv \pi_1(\pi_2(a))$ and $a.3 \equiv \pi_2(\pi_2(a))$. □

Types can be parameterised. For example, one may introduce the inductive unit types $\mathbf{1}(A, a)$: it is an inductive type with only one object $*(A, a)$ and parameterised by a type A and an object a of type A. (See Section 2.2.1 for more details on the parameterised unit types.)

One may also introduce type universes to collect (the names of) some types into types [36]. This can be considered as a reflection principle: such a universe reflects those types whose names are its objects. For instance, in Martin-Löf's type theory or UTT, we can introduce a universe $U : Type$, together with $T : (U)Type$, to reflect the types in $Type$ introduced before U (see [36] or §9.2.3 of [25]). For example, for Σ-types, we introduce their names in U as follows

$$\frac{\Gamma \vdash a : U \quad \Gamma \vdash b : (T(a))U}{\Gamma \vdash \sigma(a,b) : U} \qquad \frac{\Gamma \vdash a : U \quad \Gamma \vdash b : (T(a))U}{\Gamma \vdash T(\sigma(a,b)) = \Sigma(T(a), [x{:}T(a)]T(b(x))) : Type}$$

Note that such a universe is predicative: for example, U and $Nat \to U$ do not have names in U.

Remark The type theories thus specified in LF are intensional type theories as implemented in the proof assistants Agda [1], Coq [10], Lego [33] and Matita [37].[3] They have nice meta-theoretic properties including Church-Rosser, Subject Reduction and Strong Normalisation. (See Goguen's thesis

[3] In some systems (Agda, for example), there may be some experimental features that are extensional, but the cores of these proof assistants are all intensional.

on the meta-theory of UTT [16].) In particular, the inductive types do not have the extensional η-like equality rules. As an example, the above inductive unit type is different from the singleton type [3] in that, for a variable $x : \mathbf{1}(A, a)$, x is not computationally equal to $*(A, a)$. □

Coercive subtyping. Coercive subtyping for dependent type theories has been developed and studied as a general approach to abbreviation and subtyping in type theories with inductive types [26, 27]. Coercions have been implemented in the proof assistants Coq [10, 42], Lego [33, 4], Plastic [7] and Matita [37]. Here, we explain the main idea and introduce necessary terminologies. For a formal presentation with complete rules, see [27].

In coercive subtyping, A is a subtype of B if there is a coercion $c : (A)B$, expressed by $\Gamma \vdash A <_c B : Type$. The main idea is reflected by the following *coercive definition rule*, expressing that an appropriate coercion can be inserted to fill up the gap in a term:

$$\frac{\Gamma \vdash f : (x{:}B)C \quad \Gamma \vdash a : A \quad \Gamma \vdash A <_c B : Type}{\Gamma \vdash f(a) = f(c(a)) : [c(a)/x]C}$$

In other words, if A is a subtype of B *via* coercion c, then any object a of type A can be regarded as (an abbreviation of) the object $c(a)$ of type B.

Coercions may be declared by the users. They must be *coherent* to be employed correctly. Essentially, coherence expresses that the coercions between any two types are unique (and that there are no coercions between the same types). Formally, given a type theory T specified in LF, a set R of coercion rules is coherent if the following rule is admissible in $T[R]_0$:[4]

$$\frac{\Gamma \vdash A <_c B : Type \quad \Gamma \vdash A <_{c'} B : Type}{\Gamma \vdash c = c' : (A)B}$$

Coherence is a crucial property. Incoherence would imply that the extension with coercive subtyping is not conservative in the sense that more judgements of the original type theory T can be derived. In most cases, coherence does imply conservativity (e.g., the proof method in [43] can be used to show this). When the employed coercions are coherent, one can always insert coercions

[4] $T[R]_0$ is an extension of T with the subtyping rules in R together with the congruence, substitution and transitivity rules for the subtyping judgements, but *without* the coercive definition rule. See [27] for formal details.

correctly into a derivation in the extension to obtain a derivation in the original type theory. For an intensional type theory, coercive subtyping is an *intensional extension*. In particular, for an intensional type theory with nice meta-theoretic properties, its extension with coercive subtyping has those nice properties, too.

Remark Coercive subtyping corresponds to the view of types as consisting of canonical objects while 'subsumptive subtyping' (the more traditional approach with the subsumption rule) to the view of type assignment [32]. These two notions of subtyping are suitable for different kinds of type systems: subsumptive subtyping for type assignment systems such as the polymorphic calculi in programming languages and coercive subtyping for the type theories with canonical objects such as Martin-Lofs type theory implemented in proof assistants. It is worth noting that subsumptive subtyping is incompatible with the idea of canonical object and cannot be employed adequately for type theories with canonical objects, while coercive subtyping can be used to do so satisfactorily [31]. Furthermore, coercive subtyping is not only suitable for structural subtyping, but for non-structural subtyping. The use in this paper of the coercion ξ concerning the unit type (see Sections 2.2.2 and 2.2.3) is such an example. □

2. Manifest Fields and Intensional Manifest Fields in Σ-types

We shall first give a brief introduction to manifest fields and how they may be introduced with some notions of extensional equality, and then consider how to introduce them for Σ-types in an intensional way by means of coercive subtyping.

2.1. Manifest Fields with Extensional Notions of Equality

Σ-types or dependent record types can be used to represent types of modules. For instance, a type of some kind of abstract algebras may be represented as (cf., the notational convention for Σ-types in Section 1.1)

$$M \equiv \sum[S : U,\ op : S \to S,\ ...],$$

where S stands for (the type of) the carrier set with U being a type universe. (For simplicity, we omit the details such as the equality over S etc.)

In the following, we shall give an example to show how manifest fields [22] in a type of modules may be used and then explain how they are traditionally associated with extensional notions of equality.

An example of manifest fields. One may use a manifest field to specify that the data expected in a field is not only of a given type, but the same as a *specific* object of that type. For instance, consider the above Σ-type M. For $m : M$, we want to define a subtype of M such that the carrier set of any object of the subtype must be the same as that of m (i.e., $m.1$ – see the notational convention in Section 1.1). This subtype of modules can be defined as

$$\sum[S = m.1 : U,\ op : S \to S,\ ...],$$

which is the same as M except that the first field is *manifest*, specifying that the data in that field must be the same as $m.1$.

Extensional manifest fields. When manifest fields are introduced in a direct way, they introduce an *extensional* notion of equality [18, 40, 12]. For instance, in the above example, (the variable) S and $m.1$ are computationally equal and, in particular, they are interchangeable in type-checking. *Extensional manifest fields* are associated with such an equality that is a strong form of η-like equality which makes the meta-theoretic studies rather difficult.

Extensional encodings of manifest fields. Manifest fields can be coded by means of other extensional constructs, including the extensional equality $I(A, a, b)$, which was first introduced in Martin-Löf's extensional type theory (ETT) [36] and adopted by NuPRL [9]. In ETT, the propositional equality $I(A, a, b)$ is equivalent to the judgemental equality:

$$\frac{\Gamma \vdash p : I(A, a, b)}{\Gamma \vdash a = b : A}$$

With this strong extensional equality, one may express a manifest field $v = a : A$ by means of the following two (abstract) fields [8]:

$$v : A, \quad x : I(A, v, a),$$

where the second guarantees that v is judgementally equal to a. As is known, because of the strength of $I(A, a, b)$, strong normalisation fails and type checking is undecidable (for a proof of the latter, see [20]).

Another extensional construct that can be used to encode manifest fields is the singleton type [3, 19, 44]. For $a : A$, b is an object of the singleton type $\{a\}_A$ if and only if b and a are judgementally equal. With this, a manifest field $v = a : A$ can simply be represented as the field $v : \{a\}_A$, because then v

is judgementally equal to a. The singleton types also introduce a strong form of η-like equality (among other things such as subtyping) and are difficult in meta-theory. (See [13] for a sophisticated proof of strong normalisation of a simple type system with singleton types.)

Remark It has been thought that it would be difficult, if not impossible, to have manifest fields in type theory without the help of some extensional notion of equality. This is partly because that, in an intensional type theory, the propositional equality is not equivalent to the computational (judgemental) equality in a non-empty context and, therefore, to express $v = a : A$, it is not enough to just have a proof that v is propositionally equal to a; we would need a way to make them judgementally equal (for example, for type-checking). However, as we shall show below, manifest fields can be expressed in an intensional type theory, with the help of coercive subtyping. □

2.2. Intensional Manifest Fields in Σ-types

We shall now introduce manifest fields in Σ-types in an intensional type theory. This is made possible by means of a special form of coercive subtyping.

2.2.1. Unit Types

An inductive unit type is one that has only one object and its induction principle says that, if one can prove that some property holds for its only object, that property holds for all of its objects (including variables). A type can be parameterised and so can a unit type. We shall use a special form of unit type:

$$\mathbf{1}(A, a),$$

which is parameterised by a type A and an object a of type A. It can be introduced formally by declaring the following constants in LF:

$$\begin{array}{lcl} \mathbf{1} & : & (A{:}Type)(x{:}A)\ Type \\ * & : & (A{:}Type)(x{:}A)\ \mathbf{1}(A, x) \\ \mathcal{E} & : & (A{:}Type)(x{:}A) \\ & & (C : (\mathbf{1}(A, x))Type)(c : C(*(A, x))(z : \mathbf{1}(A, x))C(z) \end{array}$$

with the following computation rule for the elimination operator $\mathcal{E}$:

$$\mathcal{E}(A, x, C, c, *(A, x)) = c,$$

where $A : Type$, $x : A$, $C : (\mathbf{1}(A, x))Type$ and $c : C(*(A, x))$.

Remark Inductive types in an intensional type theory do not have the η-like equality rules. For example, if variable x is of type $\mathbf{1}(A, a)$, x is not computationally equal to $*(A, a)$. This is different from the singleton type [3] where, if $x : \{a\}_A$, x is computationally equal to a. □

2.2.2. IMFs in Σ-types

An *intensional manifest field* (IMF for short) in a Σ-type is a field of the form

$$x : \mathbf{1}(A, a).$$

It will be written by means of the following notation:

$$x \sim a : A.$$

In other words, we write $\sum[... \ x \sim a : A, \ ...]$ for $\sum[... \ x : \mathbf{1}(A, a), \ ...]$.

The intention is that, for an IMF $x \sim a : A$, the variable x stands for the object a of type A. Here is the example considered in Section 2.1, except that we use IMFs rather than extensional manifest fields.

Example 2.1. *Consider the type M of algebras in Section 2.1, repeated here:*

$$M \equiv \sum[S : U, \ op : S \to S, \ ...],$$

Let $m : M$ and we want to define a subtype M_w of M to express the idea that any module of type M_w has the same carrier type as that of m. Such a subtype of modules may be expressed as follows:

$$M_w \equiv \sum[S \sim m.1 : U, \ op : S \to S, \ ...].$$

where the first field is an IMF. Intuitively, we want S to stand for the type $m.1$. In reality, however, S is an object of type $\mathbf{1}(U, m.1)$ – it is not a type! How can S be used as $m.1$ so that, for example, $S \to S$ be well-typed? It is here that coercive subtyping plays a crucial role as explained below. □

The Σ-types involving IMFs are well-defined and behave as intended with the help of the following two coercions, whose formal definitions and properties such as coherence are considered in Section 2.2.3:

- $\xi_{A,a}$ is from $\mathbf{1}(A,a)$ to A and maps every object of $\mathbf{1}(A,a)$ to a. In a context where an object of type A is required after the IMF $x \sim a : A$ (e.g., in the field $op : S \to S$ in the above Example 2.1), x is coerced into a and behaves as an abbreviation of $\xi_{A,a}(x) = a$.

- The component-wise coercions for Σ-types d^i_Σ $(i = 1, 2, 3)$ propagate subtyping relations, including those specified by ξ, through Σ-types so that the IMFs can be used properly in larger contexts.

Example 2.2. *We continue the above Example 2.1 to show how the coercion ξ is used to support IMFs. With coercion ξ, the type M_w is well-typed: $S \to S$ is well-typed because S is now coerced into the type $\xi_{U,m.1}(S) = m.1$.*[5] □

In the following, we formally define ξ and d^i_Σ, show that they are coherent together and, then, study the `with`-clauses for specifying IMFs in Σ-types.

2.2.3. Coercions ξ and Component-wise Coercions

The rules for coercion ξ and the component-wise coercions for Σ-types are given in Figure 1. The component-wise rules express the idea that subtyping relations propagate through the Σ-types: informally, if A is a subtype of A' and B is a family of subtypes of B', then $\Sigma(A, B)$ is a subtype of $\Sigma(A', B')$. For example, when

$$A <_c A' \quad \texttt{and} \quad B(x) <_{c'\{x\}} B'(c(x)),$$

we have

$$\Sigma(A, B) <_{d^3_\Sigma} \Sigma(A', B'),$$

where d^3_Σ maps (a, b) to $(c(a), c'\{a\}(b))$.

The coercion rules in Figure 1 are coherent when put together, as the following proposition shows.

Proposition 2.3 (Coherence). *Let $\mathcal{R} = \{(\xi), (d^1_\Sigma), (d^2_\Sigma), (d^3_\Sigma)\}$. Then $\mathcal{R}$ is coherent.*

Proof. By induction on derivations, we prove the more general statement:

[5] One may notice that this is an example of the so-called 'kind coercion' [4, 42], which is a special case of argument coercions in an LF formulation of coercive subtyping (see, for example, [27]).

Coercion ξ

$$(\xi) \qquad \frac{\Gamma \vdash A : Type \quad \Gamma \vdash a : A}{\Gamma \vdash \mathbf{1}(A, a) <_{\xi_{A,a}} A : Type}$$

where $\xi_{A,a} = [x : \mathbf{1}(A, a)]a$.

Component-wise coercions for Σ***-types***

$$(d^1_\Sigma) \qquad \frac{\Gamma \vdash B' : (A')Type \quad \Gamma \vdash A <_c A' : Type}{\Gamma \vdash \Sigma(A, B' \circ c) <_{d^1_\Sigma} \Sigma(A', B') : Type}$$

where $d^1_\Sigma = [z : \Sigma(A, B' \circ c)](c(\pi_1(z)), \pi_2(z))$.

$$(d^2_\Sigma) \qquad \frac{\begin{array}{c}\Gamma \vdash B : (A)Type \quad \Gamma \vdash B' : (A)Type \\ \Gamma, x{:}A \vdash B(x) <_{c'\{x\}} B'(x) : Type\end{array}}{\Gamma \vdash \Sigma(A, B) <_{d^2_\Sigma} \Sigma(A, B') : Type}$$

where $d^2_\Sigma = [z : \Sigma(A, B)](\pi_1(z), c'\{\pi_1(z)\}(\pi_2(z)))$.

$$(d^3_\Sigma) \qquad \frac{\begin{array}{c}\Gamma \vdash B : (A)Type \quad \Gamma \vdash B' : (A')Type \\ \Gamma \vdash A <_c A' : Type \quad \Gamma, x{:}A \vdash B(x) <_{c'\{x\}} B'(c(x)) : Type\end{array}}{\Gamma \vdash \Sigma(A, B) <_{d^3_\Sigma} \Sigma(A', B') : Type}$$

where $d^3_\Sigma = [z : \Sigma(A, B)](c(\pi_1(z)), c'\{\pi_1(z)\}(\pi_2(z)))$.

Figure 1: Basic subtyping rules (ξ) and (d^i_Σ)

- if $\Gamma \vdash A <_c B : Type$ and $\Gamma \vdash A' <_{c'} B' : Type$, where $\Gamma \vdash A = A' : Type$ and $\Gamma \vdash B = B' : Type$, then $\Gamma \vdash c = c' : (A)B$.

For example, in the case that the last rules to derive $A <_c B$ and $A' <_{c'} B'$ are both (ξ) with $c \equiv \xi_{C,a}$ and $c' \equiv \xi_{C',b}$, we have that $\mathbf{1}(C, a) \equiv A = A' \equiv \mathbf{1}(C', b)$. Then, by Church-Rosser, $C = C'$, $a = b$, and $\xi_{C,a}(x) = a = b = \xi_{C',b}(x)$ for any $x : \mathbf{1}(C, a)$. Therefore, $\xi_{C,a} = \xi_{C',b}$ by the LF-rules $(*)$ and (η) in Appendix A. □

2.3. ***with****-clauses*

Manifest fields can be introduced by means of the `with`-clauses (see, e.g., [40]). Usually, they introduce extensional manifest fields with new computation rules. We shall instead consider them with the intensional manifest fields.

Intuitively, given a Σ-type with a field $v : A$, a `with`-clause modifies it into the same Σ-type except that the corresponding field becomes manifest: $v \sim a : A$ (i.e., $v : \mathbf{1}(A, a)$). For instance, the module type M_w in Example 2.1 can be obtained from M as follows:

$$M_w = M \ \underline{\texttt{with}} \ \underline{\texttt{field}} \ 1 \ \underline{\texttt{as}} \ m.1.$$

Here is the definition of the `with`-clauses for Σ-types.

Definition 2.4 (`with`-clause for Σ-types). *Let* $M \equiv \sum[x_1 : A_1, \ ..., \ x_n : A_n]$, $i \in \{1, ..., n\}$ *and* $x_1 : A_1, \ ..., \ x_{i-1} : A_{i-1} \vdash a : A_i$. *Then,*

$$\begin{aligned} & M \ \underline{\texttt{with}} \ \underline{\texttt{field}} \ i \ \underline{\texttt{as}} \ (x_1, ..., x_{i-1})a \\ =_{\text{df}} & \textstyle\sum[\ x_1 : A_1, \ ..., \ x_{i-1} : A_{i-1}, \quad x_i \sim a : A_i, \quad x_{i+1} : A_{i+1}, \ ..., \ x_n : A_n \ ]. \end{aligned}$$

When $x_j \notin FV(a)$ $(j = 1, ..., i-1)$, *we omit the variables* x_j *and simply write* $(M \ \underline{\texttt{with}} \ \underline{\texttt{field}} \ i \ \underline{\texttt{as}} \ a)$ *for* $(M \ \underline{\texttt{with}} \ \underline{\texttt{field}} \ i \ \underline{\texttt{as}} \ (x_1, ..., x_{i-1})a)$. □

Remark Note that the Σ-type M in the above definition may already have intensional manifest fields, which after all are just notations for abstract fields whose types are special unit types. □

The fields in tuples (objects of Σ-types) can be modified similarly.

Definition 2.5 (|-operation for Σ-types). *Let* $M \equiv \sum[x_1 : A_1,\ ...,\ x_n : A_n]$ *and* $m : M$. *Then* $m|_i =_{\mathrm{df}} (m.1, ..., m.(i-1),\ *(A'_i, m.i),\ m.(i+1), ..., m.n)$, *where* $A'_i \equiv [m.1, ..., m.(i-1)/x_1, ..., x_{i-1}]A_i$. □

Remark It is obvious that `with`-clauses can be nested. For instance,

$$M\ \underline{\texttt{with}}\ (\underline{\texttt{field}}\ i\ \underline{\texttt{as}}\ a\ \underline{\texttt{and}}\ \underline{\texttt{field}}\ j\ \underline{\texttt{as}}\ b)$$

is

$$(M\ \underline{\texttt{with}}\ \underline{\texttt{field}}\ i\ \underline{\texttt{as}}\ a)\ \underline{\texttt{with}}\ \underline{\texttt{field}}\ j\ \underline{\texttt{as}}\ b.$$

This is similar for |-operations. E.g., $m|_{i,j}$ is $(m|_i)|_j$. □

The following proposition shows that the above definitions are adequate in the sense that they behave as intended and that, if we modify a Σ-type by a `with`-clause appropriately, we obtain a subtype and, therefore, Σ-types with IMFs can be used adequately in any context.

Proposition 2.6. *Assume that* $M \equiv \sum[x_1 : A_1,\ ...,\ x_n : A_n]$ *be a* Σ*-type.*

1. *Let* $m : M$ *and* $i \in \{1, ..., n\}$.
   (a) *If* $M_i \equiv (M\ \underline{\texttt{with}}\ \underline{\texttt{field}}\ i\ \underline{\texttt{as}}\ m.i)$ *is well-typed, then* $m|_i : M_i$.
   (b) *If* $x_1, ..., x_{i-1} \notin FV(a)$, *then we have that* $m.i = a$ *if and only if* $m|_i : (M\ \underline{\texttt{with}}\ \underline{\texttt{field}}\ i\ \underline{\texttt{as}}\ a)$.
2. *Let* $M_w \equiv (M\ \underline{\texttt{with}}\ \underline{\texttt{field}}\ i\ \underline{\texttt{as}}\ a)$ *be well-typed. Then,* $M_w \leq M$ *(i.e.,* $M_w = M$ *or* $M_w <_c M$ *for some* c*).*

Proof. (1a) is proved by induction on n, using the coercion ξ. (1b) is a corollary of (1a) and proved using the fact that, by type uniqueness, $m.i = a$ if and only if $M\ \underline{\texttt{with}}\ \underline{\texttt{field}}\ i\ \underline{\texttt{as}}\ m.i = M\ \underline{\texttt{with}}\ \underline{\texttt{field}}\ i\ \underline{\texttt{as}}\ a$. The proof of (2) uses coercions ξ and d^i_Σ. □

3. Dependent Record Types and Intensional Manifest Fields

Dependent record types are labelled Σ-types. For instance,

$$\langle n : Nat,\ v : Vect(n)\rangle$$

is the dependent record type with objects (called *records*) such as

$$\langle n = 2,\ v = [5, 6]\rangle,$$

where the dependency has to be respected: $[5, 6]$ must be of type $Vect(2)$. It can be argued that record types are more natural than Σ-types to be considered as types of modules. For example, the labels in dependent record types play a role in making a finer distinction between record types, which is useful in some of the applications where Σ-types are not suitable (see, for example, [29]).

There have been several research works on dependently-typed records, with applications to the study of module mechanisms for both programming and proof languages. Some introduced dependent record *types* (at the level of types such as Nat and Σ-types) [40] and some dependent record *kinds* (at the level of kinds of a logical framework such as $Type$) [5, 12] (cf., Footnote 1). It is worth remarking that types have a richer structure than kinds and record types are more powerful than record kinds. However, on the other hand, it is easier to study record kinds than record types in, for example, meta-theoretic studies. (See [29] for a further discussion with application examples.)

In this section, we shall give an improved formulation of dependent record types, study intensional manifest fields in record types, and illustrate their uses in expressing the module mechanism with ML-sharing.

3.1. Dependent Record Types

Formally, we formulate dependent record types as an extension of the intensional type theory such as Martin-Löf's type theory or UTT, as specified in the logical framework LF. The syntax is extended with those for record types and records:

$$\begin{array}{rcl} R & := & \langle\rangle \mid \langle R,\ l : A\rangle \\ r & := & \langle\rangle \mid \langle r,\ l = a : A\rangle \end{array}$$

where we overload $\langle\rangle$ to stand for both the empty record type and the empty record. Records are associated with two operations:

- *restriction* (or *first projection*) $[r]$ that removes the last component of record r;
- *field selection* $r.l$ that selects the field labelled by l.

The labels form a new category of symbols. For every finite set of labels L, we introduce a kind $RType[L]$, the kind of the record types whose (top-level)

labels are all in L, together with the kind $RType$ of all record types:

$$\frac{\Gamma \ valid}{\Gamma \vdash RType[L] \ kind} \quad \frac{\Gamma \ valid}{\Gamma \vdash RType \ kind}$$

These kinds obey obvious subkinding relationships:

$$\frac{\Gamma \vdash R : RType[L] \quad L \subseteq L'}{\Gamma \vdash R : RType[L']} \quad \frac{\Gamma \vdash R : RType[L]}{\Gamma \vdash R : RType} \quad \frac{\Gamma \vdash R : RType}{\Gamma \vdash R : Type}$$

In particular, they are all subkinds of $Type$. Equalities are also inherited by superkinds in the sense that, if $\Gamma \vdash k = k' : K$ and K is a subkind of K', then $\Gamma \vdash k = k' : K'$. The obvious rules are omitted.

The main inference rules for dependent record types are given in Figure 2. Note that, in record type $\langle R,\ l : A\rangle$, A is a family of types of kind $(R)Type$, indexed by the objects of R, and this is how dependency is embodied in the formulation.

There are also congruence rules for record types and their objects, as given in Figure 3. It is worth remarking that we pay special attention to the equality between record types. In particular, record types with different labels are not equal. For example, $\langle n : Nat\rangle \neq \langle n' : Nat\rangle$ if $n \neq n'$.

Notation We shall adopt the following notational conventions.

- For record types, we write

  $$\langle l_1 : A_1,\ ...,\ l_n : A_n\rangle \ \text{ for } \ \langle\langle\rangle,\ l_1 : A_1\rangle,\ ...,\ l_n : A_n\rangle$$

  and often use label occurrences and label non-occurrences to express dependency and non-dependency, respectively. For instance, we write

  $$\langle n : Nat,\ v : Vect(n)\rangle \text{ for } \langle\langle\langle\rangle, n : NAT\rangle,\ v : [x{:}\langle n : NAT\rangle]Vect(x.n)\rangle,$$

  where $NAT \equiv [\_{:}\langle\rangle]Nat$, and

  $$\langle R,\ l : Vect(2)\rangle \quad \text{for} \quad \langle R,\ l : [\_{:}R]Vect(2)\rangle.$$

- For records, we often omit the type information to write

  $$\langle r,\ l = a\rangle$$

  for

  $$\text{either } \langle r,\ l = a : [\_{:}R]A(r)\rangle \text{ or } \langle r,\ l = a : A\rangle.$$

  Such a simplification is possible thanks to coercive subtyping. A further explanation is given in Appendix B. □

Formation rules

$$\frac{\Gamma \ valid}{\Gamma \vdash \langle\rangle : RType[\emptyset]} \qquad \frac{\Gamma \vdash R : RType[L] \quad \Gamma \vdash A : (R)Type \quad l \notin L}{\Gamma \vdash \langle R,\ l : A\rangle : RType[L \cup \{l\}]}$$

Introduction rules

$$\frac{\Gamma \ valid}{\Gamma \vdash \langle\rangle : \langle\rangle} \qquad \frac{\Gamma \vdash \langle R,\ l : A\rangle : RType \quad \Gamma \vdash r : R \quad \Gamma \vdash a : A(r)}{\Gamma \vdash \langle r,\ l = a : A\rangle : \langle R,\ l : A\rangle}$$

Elimination rules

$$\frac{\Gamma \vdash r : \langle R,\ l : A\rangle}{\Gamma \vdash [r] : R} \qquad \frac{\Gamma \vdash r : \langle R,\ l : A\rangle}{\Gamma \vdash r.l : A([r])}$$

$$\frac{\Gamma \vdash r : \langle R,\ l : A\rangle \quad \Gamma \vdash [r].l' : B \quad l \neq l'}{\Gamma \vdash r.l' : B}$$

Computation rules

$$\frac{\Gamma \vdash \langle r,\ l = a : A\rangle : \langle R,\ l : A\rangle}{\Gamma \vdash [\langle r,\ l = a : A\rangle] = r : R} \qquad \frac{\Gamma \vdash \langle r,\ l = a : A\rangle : \langle R,\ l : A\rangle}{\Gamma \vdash \langle r,\ l = a : A\rangle.l = a : A(r)}$$

$$\frac{\Gamma \vdash \langle r,\ l = a : A\rangle : R \quad \Gamma \vdash r.l' : B \quad l \neq l'}{\Gamma \vdash \langle r,\ l = a : A\rangle.l' = r.l' : B}$$

Figure 2: The main inference rules for dependent record types

Congruence rules for record types

$$\frac{\Gamma\ valid}{\Gamma \vdash \langle\rangle = \langle\rangle : RType[\emptyset]}$$

$$\frac{\Gamma \vdash R = R' : RType[L] \quad \Gamma \vdash A = A' : (R)Type \quad l \notin L}{\Gamma \vdash \langle R,\ l : A\rangle = \langle R',\ l : A'\rangle : RType[L \cup \{l\}]}$$

Congruence rules for records

$$\frac{\Gamma\ valid}{\Gamma \vdash \langle\rangle = \langle\rangle : \langle\rangle}$$

$$\frac{\begin{array}{c}\Gamma \vdash R : RType[L] \quad l \notin L \\ \Gamma \vdash r = r' : R \quad \Gamma \vdash a = a' : A(r) \quad \Gamma \vdash A = A' : (R)Type\end{array}}{\Gamma \vdash \langle r,\ l = a : A\rangle = \langle r',\ l = a' : A'\rangle : \langle R,\ l : A\rangle}$$

$$\frac{\Gamma \vdash r = r' : \langle R, l : A\rangle}{\Gamma \vdash [r] = [r'] : R}$$

$$\frac{\Gamma \vdash r = r' : \langle R, l : A\rangle}{\Gamma \vdash r.l = r'.l : A([r])}$$

Figure 3: Congruence rules

3.2. Remarks: Meta-theoretic Properties and Beyond

Some brief remarks are necessary to explain the above formulation of dependent record types (DRTs for short).

Meta-theoretic properties. First, we mention that the above system of DRTs enjoys nice proof-theoretic properties such as Church-Rosser, Subject Reduction and Strong Normalization. Feng has developed the typed operational semantics for the DRTs in her PhD work [15, 14] and proved that the LF-extension by DRTs has those nice properties.

Record types vs. record kinds. It is important to emphasise that we have formulated record *types*, not record *kinds*. Record types are at the same level as the other types such as Nat and $A \times B$; they are not at the level of kinds such as $Type$. (For those familiar with previous work on dependently-typed records, both [5] and [12] study record *kinds* — their 'record types' are studied at the level of kinds in the logical framework, while only [40] studies record types.) Record types are much more powerful than record kinds. For example, one may introduce type universes to reflect record types, which can then be used to represent module types in a more flexible way than record kinds in many useful applications. (See [29] for a more detailed explanation with examples.) Since kinds have a much simpler structure than types, it is much easier to add record kinds to a type theory than record types. For example, a record kind must be of the form $\langle R,\ l : A\rangle$ and cannot be of other forms such as $f(k)$, but this is not the case for a record type. For example, a record type may be of the form $f(k)$, say

$$([x{:}Type]x)(\langle n : Nat,\ v : Vect(n)\rangle)$$

that is equal to $\langle n : Nat,\ v : Vect(n)\rangle$. As a consequence, it is much easier to study (e.g., to formulate) record kinds. The issue of label distinctness discussed below is an example.

Label distinctness in record types. It is easy to ensure that the labels in a record kind are distinct (as in, e.g., [12]), but when considering record types, how can one ensure that the (top-level) labels in a record type are distinct? Thinking of this carefully, one would find that it is not clear how it could be done in a straightforward way.[6] It is probably because of this difficulty

[6]There is a problem in [5], where the freshness condition of label occurrence in a formation rule of record kinds has not been clearly defined – its definition is not easy, if

that, when record types are studied in [40], a special strategy called 'label shadowing' is adopted; that is, label repetition is allowed and, if two labels are the same, the latter 'shadows' the earlier. For example, for $r \equiv \langle n = 3,\ n = 5\rangle$, $r.n$ is equal to 5 but not 3. This, however, is not natural and may cause problems in some applications.

In our formulation of dependent record types, we have introduced the kinds $RType[L]$ of record types whose (top-level) labels occur in L. This has solved the problem of ensuring label distinctness in a satisfactory way. (See the second formation rule in Figure 2.)

Independence on subtyping. Many previous formulations of dependently-typed records make essential use of subtyping in typing selection terms [18, 5, 12]. In this respect, [40] is different and our formulation follows it in that it is *independent* of subtyping. We consider this independence as a significant advantage, mainly because it allows one to adopt more flexible subtyping relations in formalisation and modelling.

As a final remark, we should mention that, in the context of extensional type theory, people have studied encodings of record types by means of other constructs. For example, in NuPRL, 'very dependent function types' and intersection types have been studied to encode dependent record types [8, 2]. However, it is difficult to see how this can be done in intensional type theories.

3.3. Intensional Manifest Fields in Record Types

Intensional manifest fields can be defined for record types similarly as we did for Σ-types in Section 2.2. In record types, for $A : (R)Type$ and $a : (r{:}R)A(r)$,

$$l \sim a : A \ \text{ stands for } \ l : [r{:}R]\mathbf{1}(A(r), a(r)).$$

In the simpler situation, for $A : Type$ and $a : A$, $l \sim a : A$ stands for $l : \mathbf{1}(A, a)$. In records, for $b : B$,

$$l \sim_B b \ \text{ stands for } \ l = *(B, b).$$

---

possible at all, because there are functional terms that result in record kinds as values. This is similar to the problem with record types.

Example 3.1. *Let* $R = \langle S : U,\ op : S \to S \rangle$ *and* $r : R$. *The* S*-field of the following record type*

$$R_w \equiv \langle S \sim r.S : U,\ op : S \to S \rangle$$

is manifest. Intuitively, it insists that, for any record of type R_w*, its* S*-field must be the same as the* S*-field of* r. □

The IMFs in record types work as intended under the assumption that we have the following two coercions:

- The coercion ξ as defined in Figure 1 in Section 2.2.3.
- The component-wise coercions d^i_R $(i = 1, 2, 3)$ for dependent record types as given in Figure 4. These coercions propagate subtyping through record types. For instance, in (d^3_R), d^3_R maps $\langle r,\ l = a \rangle$ to $\langle c(r),\ l = c'\{r\}(a) \rangle$.

Remark Note that a component-wise coercion only exists between the record types that have the same corresponding labels. For example, if $l \neq l'$, there is no component-wise coercion between $\langle l : A \rangle$ and $\langle l' : B \rangle$ even if we have $A <_c B$. □

The coercion rule (ξ) and the component-wise rules in Figure 4 are coherent together, as the following proposition shows. (See [14] for a detailed proof.)

Proposition 3.2 (Coherence). *Let* $\mathcal{R} = \{(\xi), (d^1_R), (d^2_R), (d^3_R)\}$. *Then* $\mathcal{R}$ *is coherent.* □

Note that different applications employ different coercions and, thanks to the independence of the formulation of record types with subtyping, it is flexible to use different coercions. For example, in modelling classes in an object-oriented style, as illustrated in [30], we also employ record projections as coercions, with the following rules:

$$\frac{\Gamma \vdash \langle R,\ l : A \rangle : RType}{\Gamma \vdash \langle R,\ l : A \rangle <_{[\_]} R : RType} \qquad \frac{\Gamma \vdash A : Type \quad \Gamma \vdash \langle R,\ l : A \rangle : RType}{\Gamma \vdash \langle R,\ l : A \rangle <_{Snd} \langle l : A \rangle : RType}$$

$$(d^1_R) \qquad \frac{\Gamma \vdash R : RType[L] \quad \Gamma \vdash R' : RType[L] \quad \Gamma \vdash A' : (R')Type \quad \Gamma \vdash R <_c R' : RType}{\Gamma \vdash \langle R,\ l : A' \circ c\rangle <_{d^1_R} \langle R',\ l : A'\rangle : RType} \quad (l \notin L)$$

where $d^1_R = [x : \langle R,\ l : A\rangle]\langle c([x]),\ l = x.l\rangle$.

$$(d^2_R) \qquad \frac{\Gamma \vdash R : RType[L] \quad \Gamma \vdash A,\ A' : (R)Type \quad \Gamma, x{:}R \vdash A(x) <_{c'\{x\}} A'(x) : Type}{\Gamma \vdash \langle R,\ l : A\rangle <_{d^2_R} \langle R,\ l : A'\rangle : RType} \quad (l \notin L)$$

where $d^2_R = [x : \langle R,\ l : A\rangle]\langle [x],\ l = c'\{[x]\}(x.l)\rangle$.

$$(d^3_R) \qquad \frac{\begin{array}{c}\Gamma \vdash R : RType[L] \quad \Gamma \vdash R' : RType[L] \\ \Gamma \vdash A : (R)Type \quad \Gamma \vdash A' : (R')Type \\ \Gamma \vdash R <_c R' : RType \quad \Gamma, x{:}R \vdash A(x) <_{c'\{x\}} A'(c(x)) : Type\end{array}}{\Gamma \vdash \langle R,\ l : A\rangle <_{d^3_R} \langle R',\ l : A'\rangle : RType} \quad (l \notin L)$$

where $d^3_R = [x : \langle R,\ l : A\rangle]\langle c([x]),\ l = c'\{[x]\}(x.l)\rangle$.

Figure 4: Component-wise coercions for dependent record types

where, in the second rule, A is a type, $\langle R,\ l : A\rangle$ stands for $\langle R,\ l : [\_{:}R]A\rangle$, and the kind of the second projection Snd is the non-dependent kind $(\langle R,\ l : A\rangle)\langle l : A\rangle$.[7]

Remark It is worth remarking that the labels in record types play an important role in distinguishing between record types and this can be explained by means of the coherence issue of projections. As is known from Y. Luo's thesis [23] that the projections together are not coherent coercions for Σ-types. However, the coercions $[\_]$ and Snd together are coherent. Also, if one allowed label repetitions in record types, as in [40], the projection coercions $[\_]$ and Snd would be incoherent together. □

As for Σ-types, we can modify a record type by means of a `with`-clause. For $R \equiv \langle l_1 : A_1, ...,\ l_n : A_n\rangle$, $i \in \{1, ..., n\}$ and $a : (x : R_{i-1})A_i(x)$, where $R_{i-1} \equiv \langle l_1 : A_1, ...,\ l_{i-1} : A_{i-1}\rangle$,

$$\begin{array}{rl} & R\ \underline{\texttt{with}}\ l_i\ \underline{\texttt{as}}\ a \\ =_{\mathrm{df}} & \langle\ l_1 : A_1,\ ...,\ l_{i-1} : A_{i-1},\ \ l_i \sim a : A_i,\ \ l_{i+1} : A_{i+1},\ ...,\ l_n : A_n\ \rangle. \end{array}$$

And, for $r : R$,

$$r|_{l_i} =_{\mathrm{df}} \langle l_1 = r.l_1, ..., l_{i-1} = r.l_{i-1},\ l_i \sim_{A_i(r_{i-1})} r.l_i,\ l_{i+1} = r.l_{i+1}, ..., l_n = r.l_n\rangle,$$

where $r_{i-1} \equiv \langle l_1 = r.l_1,\ ...,\ l_{i-1} = r.l_{i-1}\rangle$.

Remark Similar to Proposition 2.6, one can show that the above definitions are adequate. It is also easy to see that the `with`-clauses and the $|$-operations can be nested. □

3.4. ML-style Sharing in Intensional Type Theory

Dependent record types with intensional manifest fields are useful in various applications. In this subsection, we use record types to show how to

[7] In general, $Snd : (r{:}\langle R,\ l : A\rangle)\langle l : A([r])\rangle$ maps r to $\langle l = r.l\rangle$. First, note that the kind of Snd is different from that of field selection: the codomain of Snd is $\langle l : A([r])\rangle$, rather than simply $A([r])$. This makes an important difference: Snd is coherent with the first projection and the component-wise coercion, while field selection is not. Secondly, only non-dependent coercions (and, in this case, the non-dependent second projection) are studied in this paper. (*Dependent coercions*, where the codomain of a coercion may depend on its argument, are studied in [34].)

use the module types with intensional manifest fields to capture ML-style sharing [35, 22].[8]

In the language design for programming and formalisation, an important topic has been the development of a suitable and powerful module mechanism. Modules that support structure sharing have been of particular interest. For example, one may want to share a point of a circle and a point of a rectangle in developing a facility for bit-mapped graphics or to share the carrier set of a semigroup and that of an abelian group when constructing rings in a formal development of abstract mathematics.

For functional programming languages, two approaches to sharing have been studied: one is *sharing by parameterisation* or the Pebble-style sharing [6, 21] and the other *sharing by equations* or the ML-style sharing [35, 38]. Both have been studied in the context of formalisation of mathematics as well, especially in designing and using type theory based proof assistants.

It is known that ML-style sharing cannot be captured in an intensional type theory by the propositional equality, since it is not equivalent to the computational equality in a non-empty context [24]. Contrary to a common belief that one needs some extensional notion of equality to express ML-style sharing in type theory, it can be captured using the IMFs that are studied in this paper, as the following example illustrates.

Example 3.3. *A ring R is composed of an abelian group $(R, +)$ and a semigroup $(R, *)$, with extra distributive laws. One can construct a ring from an abelian group and a semigroup. When doing this, one must make sure that the abelian group and the semigroup share the same carrier set. One of the ways to specify such sharing is to use an 'equation' to indicate that the carrier sets are the same. This example shows that this can be done by means of the IMFs.*

The signature types of abelian groups, semigroups and rings can be represented as the following record types, respectively, where U is a type universe:

$$\begin{aligned} AG &\equiv \langle A : U,\ + : A \to A \to A,\ 0 : A,\ inv : A \to A \rangle \\ SG &\equiv \langle B : U,\ * : B \to B \to B \rangle \\ Ring &\equiv \langle C : U,\ + : C \to C \to C,\ 0 : C,\ inv : C \to C,\ * : C \to C \to C \rangle \end{aligned}$$

[8] Another application is to use the record types and IMFs to model OO-like programs and conduct verification in proof assistants. This application has been sketched in [30] and further experiments have been done in [17].

Note that an abelian group and a semigroup do not have to share their carrier sets. In order to make the sharing happen, we introduce the following record type, which is parameterised by an AG-signature and defined by means of a `with`*-clause that specifies an IMF to ensure the sharing of the carrier sets:*

$$\begin{aligned} SGw(ag) &= SG\ \underline{\texttt{with}}\ B\ \underline{\texttt{as}}\ ag.A \\ &= \langle B \sim ag.A : U,\ * : B \to B \to B \rangle, \end{aligned}$$

where $ag : AG$*. Then, the function that generates the Ring-signature from those of abelian groups and semigroups can now be defined as:*

$$\begin{aligned} & ringGen(ag, sg) \\ =_{df}\ & \langle C = ag.A,\ + = ag.+,\ 0 = ag.0,\ inv = ag.inv,\ * = sg.* \rangle, \end{aligned}$$

where the arguments $ag : AG$ *and* $sg : SGw(ag)$ *share their carrier set.* □

Experiments in proof assistants. Experiments on the above application have been done in the proof assistants Plastic [7] and Coq [10], both supporting the use of coercions. In Plastic, one can define parameterised coercions such as ξ and coercion rules for the structural coercions: we only have to declare ξ and the component-wise coercions, then Plastic obtains automatically all of the derivable coercions, as intended. However, Plastic does not support record types; so Σ-types were used for our experiments in Plastic, at the risk of incoherence of the coercions!

Coq supports a macro for dependent record types[9] and a limited form of coercions. In Coq, we have to use the identity $ID(A) = A$ on types to force Coq to accept the coercion ξ and to use type-casting as a trick to make it happen. Also, since Coq does not support user-defined coercion rules, we cannot implement the rules for the component-wise coercions; instead, we have to specify its effects on the record types individually. The Coq code for the Ring example in Example 3.3 can be found in Appendix C.

4. Conclusion

We have shown that manifest fields can be expressed in intensional type theories with the help of coercive subtyping. The intensional manifest fields

[9]It is a macro in the sense that dependent record types are actually implemented as inductive types with labels defined as global names (and, therefore, the labels of different 'record types' must be different). Coq [10] also supports a preliminary form of 'manifest fields' by means of the let-construct, which we do not use in our experiments.

strengthen the module types such as record types and provide higher-order module mechanisms for modular development of proofs and dependently-typed programs.

After developing the IMFs as reported in [30], the author became aware of the work by Sacerdoti-Coen and Tassi [41] who, in studying formalisation of mathematical structures, have attempted to represent R $\underline{\texttt{with}}$ l $\underline{\texttt{as}}$ a by means of $\Sigma r : R.\ (r.l = a)$, where $=$ is the Leibniz equality, and to employ the so-called 'manifesting coercions' in order to approximate manifest fields. We remark that using an equality relation in this way is not completely satisfactory and seems unnecessarily complicated. Our notion of IMF is simple and, coupled with the record types as formulated in this paper, provides us a powerful tool in intensional type theory.

The dependent record types studied in this paper are *intensional* in the sense that we do not have the following extensional equality rules [5, 30], which say that two records are computationally equal if their components are:

$$\frac{\Gamma \vdash r : \langle\rangle}{\Gamma \vdash r = \langle\rangle : \langle\rangle} \qquad \frac{\Gamma \vdash r : \langle R,\ l : A\rangle \quad \Gamma \vdash r' : \langle R,\ l : A\rangle \quad \Gamma \vdash [r] = [r'] : R \quad \Gamma \vdash r.l = r'.l : A([r])}{\Gamma \vdash r = r' : \langle R,\ l : A\rangle}$$

For instance, from the second rule above, we would have $\langle r,\ l = r.l\rangle = r$ for any r of type $\langle R,\ l : A\rangle$. It is unclear whether the above extensionality rules are essentially useful. It may also be interesting to conduct research to see whether the approach of typed operational semantics (TOS) as reported in [15, 14] can be extended for such extensional record types. It would be obviously problematic if one considered the reduction relation for the records as follows:

$$\langle r,\ l = r.l\rangle \rhd r$$

for, together with the η-reduction for λ-terms, the Church-Rosser property would fail to hold. A question arises here: would it possible if one takes the TOS-approach by considering a reduction relation that treats η-long normal forms (e.g., by taking the above reduction in the other direction)? This involves the development of the TOS-approach to incorporate η-long normal forms and research is needed to see whether it is possible.

Acknowledgement I am grateful to Robin Adams who, among other things, has suggested the phrase 'intensional manifest field' to me.

Appendix A. The inference rules of LF [25]

Contexts, assumptions and equality rules

$$\frac{}{\langle\rangle\ valid} \quad \frac{\Gamma \vdash K\ kind \quad x \notin FV(\Gamma)}{\Gamma, x{:}K\ valid} \quad \frac{\Gamma, x{:}K, \Gamma'\ valid}{\Gamma, x{:}K, \Gamma' \vdash x : K}$$

$$\frac{\Gamma \vdash K\ kind}{\Gamma \vdash K = K} \quad \frac{\Gamma \vdash K = K'}{\Gamma \vdash K' = K} \quad \frac{\Gamma \vdash K = K' \quad \Gamma \vdash K' = K''}{\Gamma \vdash K = K''}$$

$$\frac{\Gamma \vdash k : K}{\Gamma \vdash k = k : K} \quad \frac{\Gamma \vdash k = k' : K}{\Gamma \vdash k' = k : K} \quad \frac{\Gamma \vdash k = k' : K \quad \Gamma \vdash k' = k'' : K}{\Gamma \vdash k = k'' : K}$$

$$\frac{\Gamma \vdash k : K \quad \Gamma \vdash K = K'}{\Gamma \vdash k : K'} \quad \frac{\Gamma \vdash k = k' : K \quad \Gamma \vdash K = K'}{\Gamma \vdash k = k' : K'}$$

The kind Type

$$\frac{\Gamma\ valid}{\Gamma \vdash Type\ kind} \quad \frac{\Gamma \vdash A : Type}{\Gamma \vdash El(A)\ kind} \quad \frac{\Gamma \vdash A = B : Type}{\Gamma \vdash El(A) = El(B)}$$

Dependent product kinds

$$\frac{\Gamma \vdash K\ kind \quad \Gamma, x{:}K \vdash K'\ kind}{\Gamma \vdash (x{:}K)K'\ kind} \quad \frac{\Gamma \vdash K_1 = K_2 \quad \Gamma, x{:}K_1 \vdash K_1' = K_2'}{\Gamma \vdash (x{:}K_1)K_1' = (x{:}K_2)K_2'}$$

$$\frac{\Gamma, x{:}K \vdash k : K'}{\Gamma \vdash [x{:}K]k : (x{:}K)K'} \quad (*)\ \frac{\Gamma \vdash K_1 = K_2 \quad \Gamma, x{:}K_1 \vdash k_1 = k_2 : K}{\Gamma \vdash [x{:}K_1]k_1 = [x{:}K_2]k_2 : (x{:}K_1)K}$$

$$\frac{\Gamma \vdash f : (x{:}K)K' \quad \Gamma \vdash k : K}{\Gamma \vdash f(k) : [k/x]K'} \quad \frac{\Gamma \vdash f = f' : (x{:}K)K' \quad \Gamma \vdash k_1 = k_2 : K}{\Gamma \vdash f(k_1) = f'(k_2) : [k_1/x]K'}$$

$$\frac{\Gamma, x{:}K \vdash k' : K' \quad \Gamma \vdash k : K}{\Gamma \vdash ([x{:}K]k')(k) = [k/x]k' : [k/x]K'} \quad (\eta)\ \frac{\Gamma \vdash f : (x{:}K)K' \quad x \notin FV(f)}{\Gamma \vdash [x{:}K]f(x) = f : (x{:}K)K'}$$

Appendix B. Untyped notations for pairs and records

Coercive subtyping can be used to explain and facilitate overloading [27, 4, 28]. The adoption of the untyped notations for pairs and records is a typical example.

Formally, the notations for pairs in Martin-Löf's type theory or UTT and for records in Section 3.1 are fully annotated with type information: they are of the 'typed' forms $pair(A, B, a, b)$ and $\langle r,\ l = a : A\rangle$, rather than the 'untyped' (a, b) and $\langle r,\ l = a\rangle$, respectively. A reason for this is that, in a dependent type theory, a pair or a record without type information may have two or more incompatible types. For example, the record $\langle n = 2,\ v = [5, 6]\rangle$ has both $\langle n : Nat,\ v : Vect(2)\rangle$ and $\langle n : Nat,\ v : Vect(n)\rangle$ as its types. The

presence of the type information in the typed forms allows a straightforward algorithm for type-checking, but it is clumsy and impractical.

Can we use the simpler untyped notations instead? The answer is yes: this can be done with the help of coercive subtyping. We illustrate it for records (see Section 5.4 of [27] for a treatment of pairs). Let r' be the 'intended typed version' of r. We want to use $\langle r,\ l = a\rangle$ to stand for either of the following records:

$$\begin{array}{lcl} r_1 & \equiv & \langle r',\ l = a : [\_{:}R]A(r')\rangle\ :\ \langle R,\ l : [\_{:}R]A(r')\rangle \\ r_2 & \equiv & \langle r',\ l = a : A\rangle\ :\ \langle R,\ l : A\rangle \end{array}$$

and to be able to decide which it stands for in the context. This can be done as follows. Let L be any finite set of labels such that $l \notin L$. Consider the family

$$U_L : (R : RType[L])(A : (R)Type)(x : R)(a : A(x))Type$$

of inductive unit types $U_L(R, A, x, a)$ with the only object $u_L(R, A, x, a)$. We then declare coercions δ_1^L and δ_2^L:

$$\begin{array}{l} U_L(R, A, x, a) <_{\delta_1^L} \langle R,\ l : [\_{:}R]A(x)\rangle \\ U_L(R, A, x, a) <_{\delta_2^L} \langle R,\ l : A\rangle \end{array}$$

inductively defined as: $\delta_1^L(u_L(R, A, x, a)) = \langle x,\ l = a : [\_{:}R]A(x)\rangle$ and $\delta_2^L(u_L(R, A, x, a)) = \langle x,\ l = a : A\rangle$. Then the notation $\langle r,\ l = a\rangle$ can be used to denote the object $u_L(R, A, r', a)$ and, in a context, it will be coerced into the appropriate record r_1 or r_2 according to the contextual typing requirement.

Appendix C. Coq code for the Ring example

The following is the Coq code for the Ring example – the construction of rings from abelian groups and semi-groups that share the domains. Note that we have only formalised the signatures of the algebras, omitting their axiomatic parts.

```
(* The parameterised unit type -- Unit/unit for 1/* *)
Inductive Unit (A:Type)(a:A) : Type := unit : Unit A a.
(* Coercion for the unit type; Use ID as trick to define it in Coq *)
Definition ID (A:Type) : Type := A.
```

```
Coercion unit_c (A:Type)(a:A)(_:Unit A a) := a : ID A.
(* Abelian Groups, Semi-groups and Rings -- signatures only *)
Record AG : Type := mkAG
  { A : Set; plus : A->A->A; zero : A; inv : A->A }.
Record SG : Type := mkSG
  { B : Set; times : B->B->B }.
Record Ring : Type := mkRing
  { C : Set; plus' : C->C->C; zero' : C; inv' : C->C; times' : C->C->C }.
(* Domain-sharing semi-groups; type-casting to make unit_c happen in Coq *)
Record SGw (ag : AG) : Type := mkSGw
  { B' : Unit Set ag.(A); times'' : let B' := (B' : ID Set) in B'->B'->B' }.
Implicit Arguments B'.  Implicit Arguments times''.
(* function to generate rings from abelian/semi-groups with shared domain *)
Definition ringGen (ag : AG)(sg : SGw ag) : Ring :=
  mkRing ag.(A) ag.(plus) ag.(zero) ag.(inv) sg.(times'').
```

References

[1] Agda 2008, The Agda proof assistant, Available from the web page: `http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php`, 2008.

[2] S. Allen, et al., Innovations in computational type theory using Nuprl, Journal of Applied Logic 4 (2006).

[3] D. Aspinall, Subtyping with singleton types, in: CSL'94, LNCS'993.

[4] A. Bailey, The Machine-checked Literate Formalisation of Algebra in Type Theory, Ph.D. thesis, University of Manchester, 1999.

[5] G. Betarte, A. Tasistro, Extension of Martin-Löf's type theory with record types and subtyping, in: G. Sambin, J. Smith (Eds.), Twenty-five Years of Constructive Type Theory, Oxford University Press, 1998.

[6] R. Burstall, B. Lampson, Pebble, a kernel language for modules and abstract data types, Lecture Notes in Computer Science 173 (1984).

[7] P. Callaghan, Z. Luo, An implementation of LF with coercive subtyping and universes., Journal of Automated Reasoning 27 (2001) 3–27.

[8] R. Constable, J. Hickey, Nuprl's class theory and its applications, in: Foundations of Secure Computation, IOS Press, Amsterdam, 2000.

[9] R. Constable, et al., Implementing Mathematics with the NuPRL Proof Development System, Pretice-Hall, 1986.

[10] Coq 2007, The Coq Proof Assistant Reference Manual (Version 8.1), INRIA, The Coq Development Team, 2007.

[11] T. Coquand, G. Huet, The calculus of constructions, Information and Computation 76 (1988).

[12] T. Coquand, R. Pollack, M. Takeyama, A logical framework with dependently typed records, Fundamenta Informaticae 65 (2005).

[13] J. Courant, Strong normalisation with singleton types, Electronic Notes in Theoretical Computer Science 70 (2002).

[14] Y. Feng, A theory of dependent record types with structural subtyping, Ph.D. thesis, Royal Holloway, Univ. of London, 2010. (In preparation).

[15] Y. Feng, Z. Luo, Typed operational semantics for dependent record types, in: TYPES 2009. To appear.

[16] H. Goguen, A Typed Operational Semantics for Type Theory, Ph.D. thesis, University of Edinburgh, 1994.

[17] S. Han, Verification of Java Programs in Type Theory, Ph.D. thesis, Royal Holloway, Univ. of London, 2010. (In preparation).

[18] R. Harper, M. Lillibridge, A type-theoretic approach to higher-order modules with sharing, POPL'94 (1994).

[19] S. Hayashi, Singleton, union and intersection types for program extraction, Information and Computation 109 (1994) 174–210.

[20] M. Hoffman, Extensional Concepts in Intensional Type Theory, Ph.D. thesis, University of Edinburgh, 1995.

[21] B. Lampson, R. Burstall, Pebble, a kernel language for modules and abstract data types, Information and Computation 76 (1988).

[22] X. Leroy, Manifest types, modules and separate compilation, POPL'94 (1994).

[23] Y. Luo, Coherence and Transitivity in Coercive Subtyping, Ph.D. thesis, University of Durham, 2005.

[24] Z. Luo, A higher-order calculus and theory abstraction, Information and Computation 90 (1991).

[25] Z. Luo, Computation and Reasoning: A Type Theory for Computer Science, Oxford University Press, 1994.

[26] Z. Luo, Coercive subtyping in type theory, CSL'96, LNCS'1258 (1997).

[27] Z. Luo, Coercive subtyping, J of Logic and Computation 9 (1999) 105–130.

[28] Z. Luo, Coercions in a polymorphic type system, Mathematical Structures in Computer Science 18 (2008).

[29] Z. Luo, Dependent record types revisited, in: Proc. of the 1st Inter. Workshop on Modules and Libraries for Proof Assistants (MLPA'09), Montreal, volume 429 of *ACM Inter. Conf. Proceeding Series*.

[30] Z. Luo, Manifest fields and module mechanisms in intensional type theory, in: Types for Proofs and Programs, Proc. of Inter. Conf. of TYPES'08. LNCS 5497.

[31] Z. Luo, On subtyping for type theories with canonical objects, Draft paper, 2010.

[32] Z. Luo, Y. Luo, Transitivity in coercive subtyping, Information and Computation 197 (2005) 122–144.

[33] Z. Luo, R. Pollack, LEGO Proof Development System: User's Manual, LFCS Report ECS-LFCS-92-211, Dept of Computer Science, Univ of Edinburgh, 1992.

[34] Z. Luo, S. Soloviev, Dependent coercions, CTCS'99, ENTCS'29 (1999).

[35] D. MacQueen, Modules for standard ML, ACM Symp. on Lisp and Functional Programming (1984).

[36] P. Martin-Löf, Intuitionistic Type Theory, Bibliopolis, 1984.

[37] Matita 2008, The Matita proof assistant, Available from: `http://matita.cs.unibo.it/`, 2008.

[38] R. Milner, R.H. M. Tofts, D. MacQueen, The Definition of Standard ML (Revised), MIT, 1997.

[39] B. Nordström, K. Petersson, J. Smith, Programming in Martin-Löf's Type Theory: An Introduction, Oxford University Press, 1990.

[40] R. Pollack, Dependently typed records in type theory, Formal Aspects of Computing 13 (2002) 386–402.

[41] C. Sacerdoti-Coen, E. Tassi, Working with mathematical structures in type theory, in: Types for Proofs and Programs, LNCS'4941.

[42] A. Saïbi, Typing algorithm in type theory with inheritance, POPL'97 (1997).

[43] S. Soloviev, Z. Luo, Coercion completion and conservativity in coercive subtyping, Annals of Pure and Applied Logic 113 (2002) 297–322.

[44] C. Stone, R. Harper, Extensional equivalence and singleton types, ACM Trans. on Computational Logic 7 (2006).