Manifest Fields and Module Mechanisms in Intensional Type Theory

Zhaohui Luo[⋆]

Dept of Computer Science, Royal Holloway, Univ of London
Egham, Surrey TW20 0EX, UK
zhaohui@cs.rhul.ac.uk

Abstract. Manifest fields in a type of modules are shown to be expressible in intensional type theory without strong 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 not only a higher-order module mechanism with ML-style sharing, but a powerful modelling mechanism in formalisation and verification of OO-style program modules.

1 Introduction

A type of modules may be expressed in type theory as a Σ-type or a dependent record type. A field in such a type is usually *abstract* (of the form '$v : A$') in the sense that the data in 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 in that field is not only of a given type but the 'same' as some specific object of that type. Intuitively, manifest fields allow internal expressions of definitional entries and are hence very useful in expressing various powerful constructions in a type-theoretic setting. For example, 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) [31,20].

For a manifest field $v = a : A$, the 'sameness' of the data as object a may be interpreted as judgemental equality in type theory, as is done in all of the previous studies on manifest fields in type theory [16,37,13]. If so, this gives rise to an extensional notion of judgemental equality and such manifest fields may be called *extensional manifest fields.* In type theory, such extensional manifest fields may also be obtained by means of other extensional constructs such as the singleton type [5,17] and the extensional equality [32,11]. 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 lead to outright undecidability (in the case of the extensional equality).

⋆ This work is 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.

As shown in this paper, the 'sameness' in a manifest field does not have to be interpreted by means of an extensional equality. With the help of coercive subtyping [25], manifest fields are expressible in *intensional* type theories such as Martin-Löf's intensional type theory [35] and UTT [23]. The idea is very simple: for a of type A, 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 a 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* (IMF) and, to distinguish it from an extensional manifest field, we use the notation $v \sim a : A$ to stand for $v : \mathbf{1}(A, a)$.

Manifest fields may be introduced using the `with`-clause that intuitively expresses that a field is manifest rather than abstract. For both Σ-types and dependent record types, `with`-clauses can be introduced by means of IMFs and used as expected in the presence of the component-wise coercion that propagates subtyping relations. For Σ-types, it is shown that the employed coercions are coherent together and that the IMF-representation of `with`-clauses is adequate.

Our work on IMFs in record types is based on a novel formulation of dependent record types (without manifest fields), which is different from those in the previous studies [16,37,13] and has its own merits. Among other things, our formulation is independent of structural subtyping (as in [37]), allowing more flexible subtyping relations to be adopted in formalisation, and introduces kinds of record types, giving a satisfactory solution to the problem of how to ensure label distinctness in record types.

Intensional manifest fields can be used to express definitional entries and provide not only a higher-order module mechanism with ML-style sharing[1] but also a powerful modelling mechanism in formalisation and verification of OO-style programs. Using the record macro in Coq [12], we give examples to show, with IMFs, how ML-style sharing can be captured and how classes in OO-style programs can be modelled. Since intensional type theories are implemented in the current proof assistants, many of which support the use of coercions, the module mechanism supported by IMFs can also be used for modular development of proofs and dependently-typed programs.

The following subsection briefly describes the logical framework LF and coercive subtyping, establishing the notational conventions. In Section 2, we introduce manifest fields and explain how they may be expressed in extensional type theories. The IMFs in Σ-types are studied in Section 3. In Section 4, we formulate dependent record types, introduce the IMFs in record types, and illustrate their uses in expressing the module mechanism with ML-style sharing and in modelling OO-style classes in formalisation and verification. Some of the related and future work is discussed in the conclusion.

[1] Historically, expressing ML-style sharing is the main motivation behind the studies of manifest fields [16,20,37]. In fact, it has long been 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.

1.1 The Logical Framework and Coercive Subtyping

The Logical Framework LF. LF [23] is the typed version of Martin-Löf's logical framework [35]. It is a dependent type system for specifying type theories such as Martin-Löf's intensional type theory [35] and the Unifying Theory of dependent Types (UTT) [23]. The types in LF are called *kinds*, including:

– $Type$ – 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 Chapter 9 of [23]. We sometimes use $M[x]$ to indicate that x may occur free in M and subsequently write $M[a]$ for the substitution $[a/x]M$.

When a type theory is specified in LF, its types are declared, together with their introduction/elimination operators and the associated computation rules. Examples 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 $\Pi(A, B)$ of functions $\lambda(x{:}A)b$ and Σ-types $\Sigma(A, B)$ of dependent pairs (a, b).[2] 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.

Notation. We shall use $\sum[x_1 : A_1,\ x_2 : A_2,\ ...,\ x_n : A_n]$ to stand for $\Sigma x_1 : A_1 \Sigma x_2 : A_2\ ...\ \Sigma x_{n-1} : A_{n-1}.\ A_n$, where $n \geq 1$. (When $n = 1$, $\sum[x{:}A] =_{\mathrm{df}} A$.) Similarly, $(a_1,\ a_2,\ ...,\ a_n)$ stands for $(a_1,\ (a_2,\ ...,\ (a_{n-1}, a_n)...))$. Furthermore, for any a of type $\sum[x_1 : A_1,\ x_2 : A_2,\ ...,\ x_n : A_n]$,

– $a.i =_{\mathrm{df}} \pi_1(\pi_2(...\pi_2(\pi_2(a))...))$, where π_2 occurs $i - 1$ times $(1 \leq i < n)$, and
– $a.n =_{\mathrm{df}} \pi_2(...\pi_2(\pi_2(a))...)$, 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, the unit type $\mathbf{1}(A, x)$ is parameterised by $A : Type$ and $x : A$ and can be formally introduced by declaring:

$$\mathbf{1} : (A{:}Type)(x{:}A)\ Type$$
$$* : (A{:}Type)(x{:}A)\ \mathbf{1}(A, x)$$
$$\mathcal{E} : (A{:}Type)(x{:}A)\ (C : (\mathbf{1}(A, x))Type)(c : C(*(A, x))(z : \mathbf{1}(A, x))C(z)$$

with the computation rule $\mathcal{E}(A, x, C, c, *(A, x)) = c$.

Remark 1. The type theories thus specified are intensional type theories as those implemented in the proof assistants Agda [3], Coq [12], Lego [29] and Matita [33]. They have nice meta-theoretic properties including Church-Rosser, Subject Reduction and Strong Normalisation. (See Goguen's thesis on the meta-theory of UTT [15].) In particular, the inductive types do not have the η-like equality rules. As an example, the above unit type is different from the singleton type [5] in that, for a variable $x : \mathbf{1}(A, a)$, x is not computationally equal to $*(A, a)$. □

[2] We use $A \to B$ and $A \times B$ for the non-dependent Π-type and Σ-type, respectively. Also, see Appendix A for a further explanation for the notation of untyped pairs.

Coercive Subtyping. Coercive subtyping for dependent type theories was first considered in [2] for overloading and has been developed and studied as a general approach to abbreviation and subtyping in type theories with inductive types [24,25]. Coercions have been implemented in the proof assistants Coq [12,39], Lego [29,6], Plastic [9] and Matita [33]. Here, we explain the main idea and introduce necessary terminologies. For a formal presentation with complete rules, see [25].

In coercive subtyping, A is a subtype of B if there is a coercion $c : (A)B$, expressed by $\Gamma \vdash A \leq_c B : \mathit{Type}$.[3] 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:

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

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. 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]

$$\frac{\Gamma \vdash A \leq_c B : \mathit{Type} \quad \Gamma \vdash A \leq_{c'} B : \mathit{Type}}{\Gamma \vdash c = c' : (A)B}$$

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 [40] can be used to show this). When the employed coercions are coherent, one can always insert coercions 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 2. 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 [28]. Coercive subtyping can be introduced for inductive types in a natural way [28,27], but this would be difficult, if not impossible, for subsumptive subtyping. Furthermore, coercive

[3] In this paper, we use $\leq_c$, rather than the strict relation $<_c$, for coercion judgements and assume that the identity is always a coercion: if $\Gamma \vdash A : \mathit{Type}$, then $\Gamma \vdash A \leq_{id_A} A : \mathit{Type}$, where $id_A \equiv [x{:}A]x$. This does not make an essential difference but simplifies the component-wise coercion rules in Sections 3.1 and 4.2.

[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 [25] for formal details.

subtyping is not only suitable for structural subtyping, but for non-structural subtyping. The use in this paper of the coercion concerning the unit type is such an example. □

2 Manifest Fields via Extensional Constructs

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

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

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.) Sometimes, one may use *manifest fields* [20] to specify that the data in a field is not only of a given type, but a *specific* object of that type. For instance, for $m : M$, we want to define a subtype of M the carrier set of whose objects must be the same as that of m (i.e., $m.1$ – see Section 1.1). This module type 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 $m.1$.

Traditionally, when manifest fields are considered, they introduce an *extensional* notion of equality: in the above example, (the variable) S and $m.1$ are judgementally equal and, in particular, they are interchangeable in type-checking. Such *extensional manifest fields* can be introduced directly [16,37,13] and the associated notion of equality is a strong form of η-like equality which makes the meta-theoretic studies rather difficult.

Manifest fields can be coded by means of other extensional constructs, including the extensional equality Eq, which was first introduced in Martin-Löf's extensional type theory (ETT) [32] and adopted by NuPRL [10]. In ETT, the propositional equality $Eq(A, a, b)$ is equivalent to the judgemental equality: $\Gamma \vdash p : Eq(A, a, b)$ if and only if $\Gamma \vdash a = b : A$. With Eq, one may express a manifest field $v = a : A$ with two fields: '$v : A,\ x : Eq(A, v, a)$', where the second guarantees that v is judgementally equal to a [11]. As is known, because of the strength of Eq, the judgemental equality and type checking in ETT are undecidable.

Another extensional construct that can be used to express manifest fields is the singleton type [5,17]. For $a : A$, M is an object of the singleton type $\{a\}_A$ if and only if M and a are judgementally equal. With this, a manifest field $v = a : A$ can simply be represented as the field $v : \{a\}_A$. 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 [14] for a sophisticated proof of strong normalisation of a simple type system with singleton types.)

It has been thought that it would be difficult, if not impossible, to have manifest fields in type theory without such extensional constructs. 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, we shall show that manifest fields can be expressed in an intensional type theory, with the help of coercive subtyping.

3 Intensional Manifest Fields in Σ-Types

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

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

where $\mathbf{1}(A, a)$ is the unit type parameterised by $A : Type$ and $a : A$. 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 IMFs (and the Σ-types involved) are well-defined and behave as intended with the help of the following two coercions:

- $\xi_{A,a}$, associated with $\mathbf{1}(A, a)$, maps the objects of $\mathbf{1}(A, a)$ to a. In a context where an object of type A is required (e.g., in the Σ-type but after the field $v \sim a : A$), v is coerced into a and behaves as an abbreviation of a.
- The component-wise coercion d_Σ propagates subtyping relations, including those specified by ξ, through Σ-types so that the IMFs can be used properly in larger contexts.

Example 1. Here is an example of how the coercion ξ is used to support IMFs. Consider the module type M in Section 2, repeated here:

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

For $m : M$, we can change its first field into an IMF by specifying that the carrier set must be 'the same' as (or, more precisely, abbreviate) the carrier set of m:

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

Note that S is now of type $\mathbf{1}(U, m.1)$ and is not a type. The reason that $S \to S$ is well-typed is that S is now coerced into the type $\xi_{U,m.1}(S) = m.1$. □

3.1 Coercions ξ and d_Σ and Their Coherence

The coercion rule for ξ concerning the unit type is:

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

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

The component-wise coercion expresses the idea that the subtyping relations propagate through the module types. For Σ-types, if A is a subtype of A' and B is a 'subtype' of B', then $\Sigma(A,B)$ is a subtype of $\Sigma(A',B')$. Formally, this is formulated by means of the following rule:

$$(d_\Sigma)\qquad \frac{\begin{array}{c}\Gamma \vdash B : (A)Type \quad \Gamma \vdash B' : (A')Type \\ \Gamma \vdash A \leq_c A' : Type \quad \Gamma, x{:}A \vdash B(x) \leq_{c'[x]} B'(c(x)) : Type\end{array}}{\Gamma \vdash \Sigma(A,B) \leq_{d_\Sigma} \Sigma(A',B') : Type}$$

where d_Σ maps (a,b) to $(c(a), c'[a](b))$ and is formally defined as $d_\Sigma(z) = (c(\pi_1(z)), c'[\pi_1(z)](\pi_2(z)))$, for any $z : \Sigma(A,B)$.

Remark 3. In the literature, the component-wise rules for Σ-types are usually formulated by means of $<_c$, rather than $\leq_c$. Similar rules can be recovered. For instance, when $B \equiv B' \circ c$ and $c'[x] \equiv id_{B'(c(x))}$ and if we omit the coercion judgement for the identity coercion $c'[x]$, the above rule becomes

$$\frac{\Gamma \vdash B' : (A')Type \quad \Gamma \vdash A \leq_c A' : Type}{\Gamma \vdash \Sigma(A, B' \circ c) \leq_{d_1} \Sigma(A',B') : Type}$$

where d_1 maps (a,b) to $(c(a),b)$. □

Proposition 1 (Coherence). *Let $\mathcal{R} = \{(\xi), (d_\Sigma)\}$. Then $\mathcal{R}$ is coherent.*

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

– if $\Gamma \vdash A \leq_c B : Type$ and $\Gamma \vdash A' \leq_{c'} B' : Type$, 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 $A \leq_c B$ and $A' \leq_{c'} B'$ 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 ξ-rule and η-rule in LF (see Chapter 9 of [23]). □

3.2 `with`-Clauses and Properties

Manifest fields can be introduced by means of the `with`-clauses (see, e.g., [37]). 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 1 can be obtained from M as follows: $M_w = M \ \underline{\texttt{with}}\ \underline{\texttt{field}}\ 1\ \underline{\texttt{as}}\ m.1$.

Definition 1 (`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,*

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

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

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

Definition 2 ($|$-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 4. It is obvious that `with`-clauses can be nested. For instance, M <u>`with`</u> (<u>`field`</u> i <u>`as`</u> a <u>`and`</u> <u>`field`</u> j <u>`as`</u> b) is (M <u>`with`</u> <u>`field`</u> i <u>`as`</u> a) <u>`with`</u> <u>`field`</u> j <u>`as`</u> b. This is similar for $|$-operations. E.g., $m|_{i,j}$ is $(m|_i)|_j$. □

The above definitions are adequate as the following proposition shows.

Proposition 2. *Let* $M \equiv \sum[x_1 : A_1,\ ...,\ x_n : A_n]$ *and* $m : M$. *Then,*

1. *For* $i = 1, ..., n$, *if* $M_i \equiv (M$ <u>`with`</u> <u>`field`</u> i <u>`as`</u> $m.i)$ *is well-typed, then* $m|_i : M_i$, *and*
2. *If* $x_1, ..., x_{i-1} \notin FV(a)$, *then* $m.i = a$ *if and only if* $m|_i : (M$ <u>`with`</u> <u>`field`</u> i <u>`as`</u> $a)$.

Proof. (1) is proved by induction on n, using the coercion ξ. (2) is a corollary of (1) and proved using the fact that, by type uniqueness, $m.i = a$ if and only if M <u>`with`</u> <u>`field`</u> i <u>`as`</u> $m.i = M$ <u>`with`</u> <u>`field`</u> i <u>`as`</u> a. □

The following proposition shows 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 3. *Let* M *and* $M_w \equiv (M$ <u>`with`</u> <u>`field`</u> i <u>`as`</u> $a)$ *be* Σ*-types. Then,* $M_w \leq M$ *(i.e.,* $M_w \leq_c M$ *for some* c*).*

Proof. The proof uses both coercions ξ and d_Σ. □

4 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.

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

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

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

Computation rules

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

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

Fig. 1. The main inference rules for dependent record types

4.1 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 record types $\langle\rangle$ and $\langle R,\ l : A\rangle$ and records $\langle\rangle$ and $\langle r,\ l = a : A\rangle$, 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 and *field selection* $r.l$ that selects the field labelled by l.

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. We shall also introduce the kind $RType$ of all record types. These kinds obey obvious subkinding relationships:

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

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 1. Note that, in record type $\langle R,\ l : A\rangle$, A is a family of types, indexed by the objects of R, and this is how dependency is embodied in the formulation.

Notation. For record types, we write $\langle l_1 : A_1,\ ...,\ l_n : A_n\rangle$ for $\langle\langle\langle\rangle,\ l_1 : A_1\rangle,\ ...,\ l_n : A_n\rangle$ and often use label occurrences and label non-occurrences to express dependency and non-dependency, respectively. For instance, we write

$\langle n : Nat,\ v : Vect(n)\rangle$ for $\langle n : [\_{:}\langle\rangle]Nat,\ v : [x{:}\langle n : [\_{:}\langle\rangle]Nat\rangle]Vect(x.n)\rangle$ and $\langle R,\ l : Vect(2)\rangle$ for $\langle R,\ l : [\_{:}R]Vect(2)\rangle$.

For records, we often omit the type information to write $\langle r,\ l = a\rangle$ for either $\langle r,\ l = a : [\_{:}R]A(r)\rangle$ or $\langle r,\ l = a : A\rangle$. Such a simplification is available thanks to coercive subtyping. A further explanation is given in Appendix A. □

The notion of equality between records is weakly extensional in the sense that two records are equal if their components are. This is reflected in the following two rules (similar rules are used in [7]):

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

For example, for any $r : \langle R,\ l : A\rangle$ (r can be a variable), we have, by the second rule above, that $r = \langle [r],\ l = r.l : A\rangle : \langle R,\ l : A\rangle$.

There are also congruence rules for record types and their objects, which we omit here. However, 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 $n \neq n'$.

Remark 5. These remarks are mainly for people who are familiar with previous work on dependent records. First, it is worth pointing out that, as in [37], our formulation of record types is *independent* of structural subtyping; this is different from the other previous formulations [16,7,13], which have all made an essential use of structural subtyping. We consider this independence as a significant advantage, mainly because it allows one to adopt more flexible subtyping relations in formalisation and modelling. We also comment that, although it might have its own advantages (e.g., the economy in some rule formulations), mixing subtyping with dependent records is not an easy matter: it is meta-theoretically difficult and sometimes may lead to undecidability [16].

Our formulation is different from that in [37] which allows label repetitions ('label shadowing'). We have introduced kinds $RType[L]$ and this gives a satisfactory solution to the problem of how to ensure label distinctness (or to avoid label repetition) in record types. For example, this is used essentially in Appendix A when notational coercions are defined. Also, ensuring label distinctness makes it possible for us to employ structural coercions such as projections coherently in some applications such as OO-modelling discussed in Section 4.3.

Note that we have formulated record *types*, not record *kinds*. In the terminology used in this paper, both [7] and [13] study record *kinds* – their 'record types' are studied at the level of kinds in the logical framework. Since kinds have a much simpler structure than types, it is easier to add record kinds (e.g., to ensure label distinctness) than record types, while the latter is more powerful.

Finally, 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 [11,4]. However, it is difficult to see how this can be done in intensional type theories. □

4.2 Intensional Manifest Fields in Record Types

Intensional manifest fields can be defined for record types similarly as we did for Σ-types in Section 3. 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 $b : B$,

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

Example 2. For $R = \langle S : U,\ op : S \to S\rangle$ and $r : R$, the S-field of the record type $\langle S \sim r.S : U,\ op : S \to S\rangle$ is manifest; intuitively, it insists that, for any record of this type, its S-field must be the same as the S-field of r. □

The component-wise coercion d_R for record types is given by the rule

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

where d_R maps $\langle r,\ l = a\rangle$ to $\langle c(r),\ l = c'[r](a)\rangle$ and is formally defined as, for any $r' : \langle R,\ l : A\rangle$, $d_R(r') =_{\mathrm{df}} \langle c(r_0),\ l = c'[r_0](r'.l)\rangle$, where $r_0 \equiv [r']$.

Remark 6. 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 $A \leq_c B$. □

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 OO-modelling as illustrated in Section 4.3 below, we also employ the projections as coercions, with the following rules:

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

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 $(\langle R,\ l : A\rangle)\langle l : A\rangle$.[5]

Remark 7. Assuming that the extension with dependent record types has nice meta-theoretic properties such as Church-Rosser, we can show that the coercions ξ, d_R, $[\_]$ and Snd are coherent together. It is worth remarking that the

[5] 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 [30].)

labels in record types play an important role for this coherence – the projections together are not coherent coercions for Σ-types [21]. Also, if one allowed label repetitions in record types, as in [37], the projection coercions [_] and Snd would be incoherent together. $\square$

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 $R_{i-1} \equiv \langle l_1 : A_1, ..., \ l_{i-1} : A_{i-1}\rangle$,

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

And, for $r : R$,

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

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

Remark 8. Similar propositions as Propositions 2 and 3 would show that the above definitions are adequate. It is also easy to see that the `with`-clauses and the $|$-operations can be nested. $\square$

4.3 Modules and OO-Modelling in Intensional Type Theory

In this subsection, we show how to use the module types with intensional manifest fields to capture ML-style sharing [31,20] and to model classes in OO-style programs. We shall use record types in our examples.

Modules with ML-Style Sharing. In the language design for programming and formalisation, the topic of developing a suitable and powerful module mechanism has been attracting a lot of interests. A module mechanism that supports structure sharing has 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 [8,19] and the other *sharing by equations* or the ML-style sharing [31,34]. 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 [22]. Contrary to the common belief (cf., Section 2), however, ML-style sharing can be captured using the IMFs in intensional type theory, as the following example illustrates.

```
class inc_cell is                  subclass re_inc_cell of inc_cell is
  var contents : Integer;            var backup : Integer;
  method get(): Integer is           method restore() is
    return contents end;               contents := backup end;
  method set(n:Integer) is           override set(n:Integer) is
    if get() < n                       if get() < n
    then contents := n end;            then { backup := contents;
end;                                          contents := n } end;
                                   end;
```

Fig. 2. The class inc_cell and its subclass re_inc_cell

Example 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:

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

Note that an abelian group and a semigroup do not have to share their carrier sets. In order to make this 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:

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

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

Modelling OO-Style Classes. Since definitional entries can be specified by means of IMFs, record types can be used to model the modular entities like classes in an object-oriented language, where methods are modelled as IMFs.

Example 4. Consider the class `inc_cell` in Figure 2, representing a memory cell whose content only increases. `inc_cell` can be interpreted as:

$$Cell_0 = \langle\ c : S_{cell},\ get \sim f_g : T_g,\ set \sim f_s : T_s\ \rangle$$

where

- $S_{cell} \equiv \langle\ contents : Int\ \rangle$ interprets the states of `inc_cell`;
- $f_g \equiv \lambda(s{:}S_{cell})s.contents$, of type $T_g \equiv S_{cell} \rightarrow Int$, interprets `get`; and
- $f_s \equiv \lambda(n{:}Int, s{:}S_{cell})$ `if` $get(s) < n$ `then` $\langle contents = n\rangle$ `else` s, of type $T_s \equiv Int \rightarrow S_{cell} \rightarrow S_{cell}$, interprets `set`.

Note that, in f_s, get can be applied to $s : S_{cell}$ because that it is coerced into $\xi_{T_g,f_g}(get) = f_g$ of type $T_g \equiv S_{cell} \rightarrow Int$. □

The interpretation in the above example follows the basic idea in functional interpretations of OO-languages [18,36]. In particular, when a method is interpreted, the type of states (S_{cell} in the above example) is used as argument and result types to model the effect of retrieving from and modifying the memory. There is a known problem with such a way of using state types in the interpretation when subclasses are considered [1]: the contravariance of subtyping does not lead to the natural subtyping relations between the interface types. For instance, if the subclass `re_inc_cell` in Figure 2 is interpreted similarly as in Example 4, the states of `re_inc_cell` would be interpreted as $S_{recell} \equiv \langle\ contents : Int,\ backup : Int\ \rangle$ and, for example, the method `set` as a function of type $T'_s \equiv Int \rightarrow S_{recell} \rightarrow S_{recell}$. Although $S_{recell} \leq S_{cell}$ (*via* projection coercions), T'_s is not a subtype of T_s ('the problem of contravariance'). As a consequence, the interface type of `re_inc_cell` is not a subtype of that of its superclass `inc_cell`. However, this would be very desirable and the problem can be solved in our setting by introducing a notion of universal state.

The universal state Ω. Given an object-oriented program, its classes form a DAG (directed acyclic graph), where an arrow from `C` to `C'` means that `C'` is a subclass of `C`. Let `C1`, ..., `Cn` be the leaves of the DAG and their states be interpreted as types S_1, ..., S_n, respectively. Then, the *universal state* (or, more precisely, the universal type of states) Ω is defined as the following (non-dependent) record type:

$$\Omega =_{\mathrm{df}} \langle\ s_1 : S_1,\ ...,\ s_n : S_n\ \rangle.$$

Now, with Ω, a method is interpreted as a function that takes a value from Ω and, if it modifies the state, returns a value to Ω. For instance, the method `set` in `inc_cell` is interpreted as a function of type $Int \rightarrow \Omega \rightarrow \Omega$ (rather than $Int \rightarrow S_{cell} \rightarrow S_{cell}$) and `set` in `re_inc_cell` as a (different) function of the same type. Therefore, the subtyping relationships between the interface types of `inc_cell` and `re_inc_cell` are as expected.

In general, the model enjoys desirable subtyping relationships between classes and their interface types. If a class `C` is interpreted as $C = \langle\ c : S_C,\ m_1 \sim a_1 : A_1,\ ...,\ m_n \sim a_n : A_n\ \rangle$, where S_C interprets the states of `C`, its interface type `I_C` is interpreted as $I_C = \langle\ c : S_C,\ m_1 : A_1,\ ...,\ m_n : A_n\ \rangle$. Therefore, we have $C \leq_c I_C$, where the coercion c is derived from ξ and d_R. Furthermore, if `C'` is a subclass of `C`, then $S_{C'}$ is a subtype of S_C (*via* projection coercions), and: $I_{C'} \leq_c I_C$, where the coercion c is derived from the structural coercions (the projection and component-wise coercions) for record types.

Remark 9. In the model construction of OO-classes as sketched above, subtype polymorphism is correctly captured and methods are invoked according to dynamic dispatch. We omit the detailed explanations here. □

Experiments in Proof Assistants. Experiments on the above applications have been done in the proof assistants Plastic [9] and Coq [12], 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 coercion (and the projection coercions for the application of OO-modelling), 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[6] 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 rule for the component-wise coercion; instead, we have to specify its effects on the record types individually. The Coq code for the Ring example in Example 3 can be found in Appendix B.

In the proof assistants such as Coq, verification of object-oriented programs can be done based on the formalisation of the model sketched above. For instance, one can show that, for the class `re_inc_cell`, it is an invariant that the backup value is always smaller than or equal to the contents value; formally, we prove in Coq, for every method m,

$$\begin{aligned}\forall s : \Omega.\ Pre(m) \Rightarrow s.(backup) \leq s.(contents)\\ \Rightarrow S(m,s).(backup) \leq S(m,s).(contents),\end{aligned}$$

where $Pre(m)$ stands for the precondition of m, $S(m,s)$ for the resulting state obtained from executing m with the initial state s, and $s.(\_)$ is the Coq-notation for field selection. Currently, we can only do small examples in formalisation and verification, partly because the manual encoding is rather tedious (and error-prone). We are working on the automated translations that will generate the Coq models of object-oriented programs and the Coq propositions of the specifications, and this will hopefully make the whole process much easier.

5 Conclusion

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

[6] 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 [12] also supports a preliminary (but improper) form of 'manifest fields' by means of the let-construct, which we do not use in our experiments.

the module types such as record types and provide higher-order module mechanisms for modular development of proofs and dependently-typed programs and powerful representation mechanisms for, for example, formalisation and verification of OO-style programs.

Recently, it has come to our attention that, studying formalisation of mathematical structures, Sacerdoti-Coen and Tassi [38] have attempted to represent R `with` l `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 intensional manifest field is simple and desirable and, coupled with the record types as formulated in this paper, provides us a powerful tool in intensional type theory.

As to future work, we mention that our formulation of dependent record types forms a promising basis for investigations on the meta-theory of dependent record types. We also hope that the proof assistants will implement dependent record types properly so that they can be used effectively in practice.

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

References

1. Abadi, M., Cardelli, L.: A Theory of Objects. Springer, Heidelberg (1996)
2. Aczel, P.: Simple overloading for type theories (manuscript, 1994)
3. The Agda proof assistant (version 2) (2008), `http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php?`
4. Allen, S., et al.: Innovations in computational type theory using Nuprl. Journal of Applied Logic 4(4) (2006)
5. Aspinall, D.: Subtyping with singleton types. In: Pacholski, L., Tiuryn, J. (eds.) CSL 1994. LNCS, vol. 933. Springer, Heidelberg (1995)
6. Bailey, A.: The Machine-checked Literate Formalisation of Algebra in Type Theory. PhD thesis, University of Manchester (1999)
7. Betarte, G., Tasistro, A.: Extension of Martin-Löf's type theory with record types and subtyping. In: Sambin, G., Smith, J. (eds.) Twenty-five Years of Constructive Type Theory, Oxford University Press, Oxford (1998)
8. Burstall, R., Lampson, B.: Pebble, a kernel language for modules and abstract data types. In: Plotkin, G., MacQueen, D.B., Kahn, G. (eds.) Semantics of Data Types 1984. LNCS, vol. 173. Springer, Heidelberg (1984)
9. Callaghan, P., Luo, Z.: An implementation of LF with coercive subtyping and universes. Journal of Automated Reasoning 27(1), 3–27 (2001)
10. Constable, R., et al.: Implementing Mathematics with the NuPRL Proof Development System. Prentice-Hall, Englewood Cliffs (1986)
11. Constable, R., Hickey, J.: Nuprl's class theory and its applications. In: Foundations of Secure Computation. IOS Press, Amsterdam (2000)
12. The Coq Development Team. The Coq Proof Assistant Reference Manual (Version 8.1), INRIA (2007)

13. Coquand, T., Pollack, R., Takeyama, M.: A logical framework with dependently typed records. Fundamenta Informaticae 65(1-2) (2005)
14. Courant, J.: Strong normalisation with singleton types. Electronic Notes in Theoretical Computer Science 70(1) (2002)
15. Goguen, H.: A Typed Operational Semantics for Type Theory. PhD thesis, University of Edinburgh (1994)
16. Harper, R., Lillibridge, M.: A type-theoretic approach to higher-order modules with sharing. In: POPL 1994 (1994)
17. Hayashi, S.: Singleton, union and intersection types for program extraction. Information and Computation 109(1/2), 174–210 (1994)
18. Kamin, S.: Inheritance in Smalltalk-80: a denotational definition. In: POPL 1988 (1988)
19. Lampson, B., Burstall, R.: Pebble, a kernel language for modules and abstract data types. Information and Computation 76(2/3) (1988)
20. Leroy, X.: Manifest types, modules and separate compilation. In: POPL 1994 (1994)
21. Luo, Y.: Coherence and Transitivity in Coercive Subtyping. PhD thesis, University of Durham (2005)
22. Luo, Z.: A higher-order calculus and theory abstraction. Information and Computation 90(1) (1991)
23. Luo, Z.: Computation and Reasoning: A Type Theory for Computer Science. Oxford University Press, Oxford (1994)
24. Luo, Z.: Coercive subtyping in type theory. In: van Dalen, D., Bezem, M. (eds.) CSL 1996. LNCS, vol. 1258. Springer, Heidelberg (1997)
25. Luo, Z.: Coercive subtyping. J. of Logic and Computation 9(1), 105–130 (1999)
26. Luo, Z.: Coercions in a polymorphic type system. Mathematical Structures in Computer Science 18(4) (2008)
27. Luo, Z., Adams, R.: Structural subtyping for inductive types with functorial equality rules. Mathematical Structures in Computer Science 18(5) (2008)
28. Luo, Z., Luo, Y.: Transitivity in coercive subtyping. Information and Computation 197(1-2), 122–144 (2005)
29. Luo, Z., Pollack, R.: LEGO Proof Development System: User's Manual. LFCS Report ECS-LFCS-92-211, Dept. of Computer Science, Univ. of Edinburgh (1992)
30. Luo, Z., Soloviev, S.: Dependent coercions. In: CTCS 1999, ENTCS 1929 (1999)
31. MacQueen, D.: Modules for standard ML. In: ACM Symp. on Lisp and Functional Programming (1984)
32. Martin-Löf, P.: Intuitionistic Type Theory. Bibliopolis (1984)
33. The Matita proof assistant (2008), `http://matita.cs.unibo.it/`
34. Milner, R., Harper, R., Tofts, M., MacQueen, D.: The Definition of Standard ML (revised). MIT, Cambridge (1997)
35. Nordström, B., Petersson, K., Smith, J.: Programming in Martin-Löf's Type Theory: An Introduction. Oxford University Press, Oxford (1990)
36. Pierce, B.C., Turner, D.N.: Simple type-theoretic foundations for object-oriented programming. J. of Functional Programming 4(2), 207–247 (1994)
37. Pollack, R.: Dependently typed records in type theory. Formal Aspects of Computing 13, 386–402 (2002)
38. Sacerdoti-Coen, C., Tassi, E.: Working with mathematical structures in type theory. In: Miculan, M., Scagnetto, I., Honsell, F. (eds.) TYPES 2007. LNCS, vol. 4941, pp. 157–172. Springer, Heidelberg (2008)
39. Saïbi, A.: Typing algorithm in type theory with inheritance. In: POPL 1997(1997)
40. Soloviev, S., Luo, Z.: Coercion completion and conservativity in coercive subtyping. Annals of Pure and Applied Logic 113(1-3), 297–322 (2002)

A Untyped Notations for Pairs and Records

Coercive subtyping can be used to explain and facilitate overloading [25,6,26]. 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 4.1 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 [25] 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:

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

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:

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

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.

B 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'').
```