Dependent Record Types Revisited[☆]

Zhaohui Luo[1]

Department of Computer Science
Royal Holloway, University of London
Egham, Surrey TW20 0EX, U.K.
zhaohui.luo@hotmail.co.uk

Abstract

Dependently-typed records have been studied in type theory to provide, for example, module mechanisms for both programming and proof languages. In this paper, we shall conduct further studies of dependent record types and show that they provide powerful and useful tools in applications. In particular, it is shown that dependent record types are more powerful than dependent record kinds (kinds at the level of a logical framework) and sometimes they are more useful than Σ-types (types of tuples without labels). Furthermore, we shall explain how manifest fields can be expressed for record types in an intensional type theory, without introducing any extensional rules.

Keywords: dependent record types, type theory

1. Introduction

Dependently-typed records have been studied in type theory previously, including (Harper and Lillibridge, 1994; Betarte and Tasistro, 1998; Pollack, 2002; Coquand et al., 2005; Luo, 2009), with applications to the study of module mechanisms for both programming and proof languages. In this paper, further studies are conducted about dependent record types (DRTs) and their applications.

First, let us make clear that we study record *types*, not record *kinds*. In a type theory with inductive types, types include those such as Nat of natural numbers and Σ-types of dependent pairs, while kinds are at the level

[☆]Preliminary versions of some parts of the current paper appeared as the workshop papers (Luo, 2010; Feng and Luo, 2010).

[1]Partially supported by the U.K. Leverhulme research grant F/07-537/AJ.

of logical framework used to specify the type theory (e.g., the kind $Type$ of all types). In the terminology used in this paper, most of the previous work studies record kinds,[2] with (Pollack, 2002) as the only notable exception. 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 much more powerful. For example, if R is a record type, one can form the types such as $\Pi x{:}R.C$ of dependent functions with domain R and $List(R)$ of lists of records of type R, while if R is a record kind, one cannot. Also, it is possible to consider universes of record types, but not for record kinds. We shall show how such universes can be introduced so that record types provide a more powerful mechanism than record kinds in expressing module types. This is illustrated by means of an example in date refinement.

In formulating dependent record types, we introduce kinds $RType[L]$ of the record types whose (top-level) labels all occur in the label set L. The associated label sets in the kinds $RType[L]$ play a crucial role in forming record types with distinct labels (among other uses). In particular, unlike (Pollack, 2002), repetition of labels is not allowed when record types are formed or when records are introduced. Such a requirement for label distinctness is not only intuitively natural, but useful in some applications, as one of our examples shows. It is interesting to note that, in type theory, to ensure label distinctness is not easy for record types, although it is easy for record kinds.

It has been a common view held by many researchers that, because one can easily introduce Σ-types as inductive types in type theory, dependent record types are not necessary — they can always be replaced by Σ-types. In this paper, we argue that such a view is not completely justified — record types provide some additional useful means that is not available for Σ-types. As we know, the only difference between a dependent record type and a Σ-type is that the former has field labels. We show that, in some applications, labels provide a useful mechanism in a finer distinction between record types so that record types can be used adequately together with some forms of structural subtyping, while Σ-types cannot.

A field in a record type may be manifest (in the form of $v = a : A$) as well as abstract (in the form of $v : A$) (Leroy, 1994). It has been believed that, in order to have manifest fields, one need to introduce some form of extensional equality rules. In fact, this is not the case: one can have *intensional manifest*

[2]For example, both (Betarte and Tasistro, 1998) and (Coquand et al., 2005) study record kinds – their 'record types' are studied at the level of kinds in a logical framework.

fields with the help of coercive subtyping (Luo, 2009). We shall explain how this can be done for DRTs.

The meta-theoretic properties of DRTs in a logical framework are studied by means of its Typed Operational Semantics, an approach developed by Goguen in his PhD thesis (Goguen, 1994, 1999), where he has developed the TOS for the type theory UTT (Luo, 1994) and proved that UTT has the nice properties such as Church-Rosser, Subject Reduction and Strong Normalisation. A similar development was done for the DRTs to show that the logical framework with DRTs has the nice properties.

The following subsections give the background and notational conventions: briefly describing the logical framework LF in which the underlying type theory is specified (§1.1) and the framework of coercive subtyping (§1.2). Dependent record types are formulated in §2 where their basic formulation is given in §2.1 and the introduction of universes of DRTs is described in §2.2. §3 illustrates why record types are more powerful than record kinds by considering an example in data refinement, §4 explains the usefulness of labels in an adequate use of record types together with structural subtyping, and §5 describes how manifest fields can be represented for DRTs with the help of coercive subtyping. The meta-theoretic properties of DRTs in a logical framework are studied by means of its TOS in §6, followed by the conclusion with discussions of related and future work.

1.1. The Logical Framework LF

LF (Luo, 1994) is the typed version of Martin-Löf's logical framework (Nordström et al., 1990). It is a dependent type system for specifying type theories such as Martin-Löf's intensional type theory (Nordström et al., 1990), the Calculus of Constructions (CC) (Coquand and Huet, 1988) and the Unifying Theory of dependent Types (UTT) (Luo, 1994). 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 (Luo, 1994).

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.) In a non-LF notation, $\Sigma(A, B)$, for example, will be written as $\Sigma x{:}A.B(x)$.

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 §5 for an example of using the unit types.)

One may also introduce type universes to collect (the names of) some types into types (Martin-Löf, 1984). 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 (Martin-Löf, 1984) or §9.2.3 of (Luo, 1994)). 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 (Agda 2008), Coq (Coq 2007), Lego (Luo and Pollack, 1992) and Matita (Matita 2008).[3] They have

[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.

nice meta-theoretic properties including Church-Rosser, Subject Reduction and Strong Normalisation. (See Goguen's thesis on the meta-theory of UTT (Goguen, 1994).) 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 (Aspinall, 1995) in that, for a variable $x : \mathbf{1}(A, a)$, x is not computationally equal to $*(A, a)$. □

1.2. 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 (Luo, 1997, 1999). Coercions have been implemented in the proof assistants Coq (Coq 2007; Saïbi, 1997), Lego (Luo and Pollack, 1992; Bailey, 1999), Plastic (Callaghan and Luo, 2001) and Matita (Matita 2008). Here, we explain the main idea and introduce necessary terminologies. For a formal presentation with complete rules, see (Luo, 1999).

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

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

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]

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

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 (Soloviev and Luo,

[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 (Luo, 1999) for formal details.

2002) 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 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 (Luo and Luo, 2005). 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. 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 §5) is such an example. □

2. Dependent Record Types

In this section, we present the formulation of dependent record types (DRTs) and describe how universes of DRTs can be introduced.

2.1. A Formulation of Dependent Record Types

A dependent record type is a type of labelled tuples. 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 dependency has to be respected: $[5,6]$ must be of type $Vect(2)$.

Formally, we formulate dependent record types as an extension of intensional type theories such as Martin-Löf's type theory or UTT, as specified in the logical framework LF. The syntax is extended with 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 L of labels, 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 1. Note that, in record type $\langle R,\ l : A\rangle$, A is a family of types of kind $(R)Type$, indexed by the records of type 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 2. 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

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

  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

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

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

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

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

Figure 1: The main inference rules for dependent record types.

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 (Luo, 1999). A further explanation is given in **??**. □

Remarks

Several remarks are in order to explain some of the design decisions in the above formulation and to compare it with previous attempts.

Record types v.s. record kinds. It is important to emphasise that we

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 2: Congruence rules

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 (Betarte and Tasistro, 1998) and (Coquand et al., 2005) study record *kinds* — their 'record types' are studied at the level of kinds in the logical framework, while only (Pollack, 2002) studies record types.)

Record types are much more powerful than record kinds. As explained later in §2.2, we can introduce 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.

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. For instance, it is easy to ensure that the labels in a record kind are distinct (as in, e.g., (Coquand et al., 2005)), but it is not easy at all if we consider record types. Let's discuss this issue now.

Label distinctness in record types. 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.[5] It is probably because of this difficulty that, when record types are studied in (Pollack, 2002), 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 (see, for example, Remark 4 in §4).

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

[5]There is a problem in (Betarte and Tasistro, 1998), 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 possible at all, because there are functional terms that result in record kinds as values. This is similar to the problem with record types.

has solved the problem of ensuring label distinctness in a satisfactory way. (See the second formation rule in Figure 1.)

It may be worth remarking that the label sets L also play other useful roles. For example, for a label $l \notin L$, one may want to define a functional operation $Extend[l](R) =_{\rm df} \langle R,\ l : [x : R]Nat\rangle$, for all $R : RType[L]$. Without label sets, it would be difficult to see how the operations such as $Extend[l]$ could be defined. (See Appendix A of (Luo, 2009) for a practical example in which such operations are used essentially.)

Independence on subtyping. Many previous formulations of dependently-typed records make essential use of subtyping in typing selection terms (Harper and Lillibridge, 1994; Betarte and Tasistro, 1998; Coquand et al., 2005). In this respect, (Pollack, 2002) 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.

2.2. Universes of Record Types

We explain here how universes of record types can be introduced.

One may collect (the names of) some types into a type called a universe (Martin-Löf, 1984). 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 (Martin-Löf, 1984) or §9.2.3 of (Luo, 1994)). For example, for Π-types, we have

$$\frac{\Gamma \vdash a : U \quad \Gamma \vdash b : (T(a))U}{\Gamma \vdash \pi(a, b) : U}$$

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

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

Similarly, we can consider universes of dependent record types.[6] We introduce the following universes:

- $U_R[L]$ to reflect the record types in $RType[L]$ (introduced before $U_R[L]$):

$$U_R[L] : Type \ \text{ and } \ T_R[L] : (U_R[L])RType[L].$$

[6]Note that we can do this because they are record *types*, not record *kinds*.

$$\frac{\Gamma \ valid}{\Gamma \vdash \overline{\langle\rangle} : U_R[\emptyset]} \quad \frac{\Gamma \ valid}{\Gamma \vdash T_R[\emptyset](\overline{\langle\rangle}) = \langle\rangle : RType[\emptyset]}$$

$$\frac{\Gamma \vdash r : U_R[L] \quad \Gamma \vdash a : (T_R[L](r))U \quad l \notin L}{\Gamma \vdash \overline{\langle r,\ l : a\rangle} : U_R[L \cup \{l\}]}$$

$$\frac{\Gamma \vdash r : U_R[L] \quad \Gamma \vdash a : (T_R[L](r))U \quad l \notin L}{\Gamma \vdash T_R[L \cup \{l\}](\overline{\langle r,\ l : a\rangle}) = \langle T_R[L](r),\ l : [x{:}T_R[L](r)]T(a(x))\rangle : RType[L \cup \{l\}]}$$

Figure 3: Introduction of names of record types.

$$\frac{\Gamma \vdash r : U_R[L] \quad L \subseteq L'}{\Gamma \vdash r : U_R[L']} \quad \frac{\Gamma \vdash r : U_R[L] \quad L \subseteq L'}{\Gamma \vdash T_R[L'](r) = T_R[L](r) : RType[L']}$$

$$\frac{\Gamma \vdash r : U_R[L]}{\Gamma \vdash r : U_R} \quad \frac{\Gamma \vdash r : U_R[L]}{\Gamma \vdash T_R(r) = T_R[L](r) : RType}$$

$$\frac{\Gamma \vdash r : U_R}{\Gamma \vdash r : U} \quad \frac{\Gamma \vdash r : U_R}{\Gamma \vdash T(r) = T_R(r) : Type}$$

Figure 4: Subtyping between universes.

- U_R to reflect the record types in $RType$ (introduced before U_R):

$$U_R : Type \ \text{ and } \ T_R : (U_R)RType.$$

Names of the record types are introduced into the universes $U_R[L]$ by the rules in Figure 3. For example, the record type $\langle n : Nat,\ v : Vect(n)\rangle$ has a name $\overline{\langle n : nat,\ v : vect(n)\rangle}$ in $U_R[\{n, v\}]$, where nat is a name of Nat in U and $vect : (Nat)U$ maps n to a name of $Vect(n)$.

Furthermore, the universes obey the following subtyping relationship that reflects the subkinding relationship between the corresponding kinds, where $L \subseteq L'$:

$$U_R[L] \leq U_R[L'] \leq U_R \leq U.$$

The subtyping relations are given by the rules in Figure 4.

Remark Note that the universes $U_R[L]$ and U_R do not have names in U, for otherwise the universes would become impredicative and the whole system inconsistent. □

3. Dependent Record Types as Module Types

One of the primary functions of dependent record types is to represent types of modules. Because record types (but not record kinds) can be reflected in universes, as explained in §2.2, they provide a more powerful mechanism for module types than record kinds, as the example below in data refinement illustrates.

Notation For readability, we shall adopt the following two notational conventions in this section.

- We shall not distinguish types and their names in a universe. In particular, we shall abuse the notations: for example, we simply write $A \to B$ for both the function type and its name and $\langle R,\ l : A\rangle$ for both the record type and its name.

- We shall use

$$\left\{\begin{array}{lcl} l_1 & : & A_1 \\ ... & & \\ l_n & : & A_n \end{array}\right\} \text{ and } \left[\begin{array}{lcl} l_1 & = & a_1 \\ ... & & \\ l_n & = & a_n \end{array}\right]$$

  to stand for the record type $\langle l_1 : A_1,\ ...,\ l_n : A_n\rangle$ and the record $\langle l_1 = a_1,\ ...,\ l_n = a_n\rangle$, respectively. For example, the record type $\langle n : Nat,\ v : Vect(n)\rangle$ (cf., the notational conventions at the end of §2.1) and its object $\langle n = 3,\ v = [a, b, c]\rangle$ are written as

$$\left\{\begin{array}{lcl} n & : & Nat \\ v & : & Vect(n) \end{array}\right\} \text{ and } \left[\begin{array}{lcl} n & = & 3 \\ v & = & [a, b, c] \end{array}\right],$$

  respectively. □

We now consider an example to show how record types can be used to represent module types in data refinement. The general idea of specification and data refinement in type theory is set out in (Luo, 1993). In general, a specification consists of a type (e.g., a record type), called the *structure type*, and a predicate over the type. The following example is based on an example given in (Luo, 1993); the key difference is that record types, instead of Σ-types, are used to represent module types. It illustrates the traditional implementation of stacks by arrays together with pointers.

Example 3.1. *We consider a specification of stacks, a specification of arrays and an implementation of stacks by means of arrays together with pointers.*

- **Stack**(Nat), a specification of stacks of natural numbers. *Its structure type* **Str**[**Stack**(Nat)] *can be represented as the following record type:*

$$\left\{\begin{array}{lcl} Stack & : & \textbf{Setoid} \\ empty & : & Stack.Dom \\ push & : & Nat \to Stack.Dom \to Stack.Dom \\ pop & : & Stack.Dom \to Stack.Dom \end{array}\right\}$$

where

$$\textbf{Setoid} \equiv \left\{\begin{array}{lcl} Dom & : & U \\ Eq & : & Dom \to Dom \to Prop \end{array}\right\}$$

with U being the universe reflecting types in $Type$ introduced before U (see §2.2) and $Prop$ the type of logical propositions (as in UTT).

The predicate of **Stack**(Nat) *expresses the axiomatic requirements of the stack structures including, for example, that the book equality $Stack.Eq$ is a congruence relation and that, for any number n and any stack s, $pop(push(n, s))$ is equal to s (i.e., $Stack.Eq(pop(push(n, s)), s)$).*

- **Array**(Nat), a specification of arrays of natural numbers. *This can be defined similarly. Its structure type* **Str**[**Array**(Nat)] *can be represented as the following record type:*

$$\left\{\begin{array}{lcl} Array & : & \textbf{Setoid} \\ newarray & : & Array.Dom \\ assign & : & Array.Dom \to \\ & & \quad Nat \to Nat \to Array.Dom \\ access & : & Array.Dom \to Nat \to Nat \end{array}\right\}$$

where indexes are represented by natural numbers (intuitively, $assign(A, n, i)$ and $access(A, k)$ stand for $A[i] := n$ and $A[k]$, respectively). The predicate of **Array**(Nat) *expresses the axioms such as, for any array A, any number n and any indexes i and j, $access(assign(A, n, i), j)$ is equal to n, if $i = j$, and $access(A, j)$, if $i \neq j$.*

- *Now, we want to use arrays to implement stacks. We can define a refinement map*

$$\rho : \textbf{Str}[\textbf{Array}(Nat)] \to \textbf{Str}[\textbf{Stack}(Nat)]$$

*as follows: for any record r : **Str**[**Array**(Nat)], $\rho(r)$ is defined to be the record given in Figure 5. In the implementation, a stack is represented*

$$
\rho(r) = \left[\begin{array}{lcl}
Stack & = & \left[\begin{array}{lcl}
Dom & = & \left\{\begin{array}{lcl} arr & : & r.Array.Dom \\ ptr & : & Nat \end{array}\right\} \\
Eq & = & \lambda(s, s' : Dom)\ s.ptr =_{Nat} s'.ptr\ \& \\
 & & \quad \forall i : Nat.\ (i < s.ptr \Rightarrow \\
 & & \quad\quad access(s.arr,\ i) =_{Nat} access(s'.arr, i))
\end{array}\right] \\
empty & = & \left[\begin{array}{lcl} arr & = & r.newarray \\ ptr & = & 0 \end{array}\right] \\
push & = & \lambda(n : Nat, s : Stack.Dom) \\
 & & \quad \left[\begin{array}{lcl} arr & = & assign(s.arr,\ n,\ s.ptr) \\ ptr & = & s.ptr + 1 \end{array}\right] \\
pop & = & \lambda(s : Stack.Dom) \\
 & & \quad \left[\begin{array}{lcl} arr & = & s.arr \\ ptr & = & s.ptr - 1 \end{array}\right]
\end{array}\right]
$$

Figure 5: Refinement map from arrays to stacks.

by means of a record that consists of an array arr and a pointer ptr; in other words, the type of stacks is refined into the record type

$$\rho(r).Stack.Dom \equiv \left\{\begin{array}{lcl} arr & : & r.Array.Dom \\ ptr & : & Nat \end{array}\right\}.$$

Two of such stack representations s and s' are equal if their pointers are the same (i.e., $s.ptr =_{Nat} s'.ptr$, where $=_{Nat}$ is the propositional equality on Nat) and accessing both of the representing arrays with an index $i < s.ptr$ gives the same result.

We can prove that ρ as defined above is indeed a refinement map in the sense that it maps every realisation of **Array**(Nat) *to a realisation of* **Stack**(Nat)*.* □

Remark Note that, in the above example, $\rho(r).Stack.Dom$ is a record type (not a record kind) and, therefore, it can be reflected as an object in a type universe. This is why the refinement map ρ is well-typed: for example, the record type $\rho(r).Stack.Dom$ has a name in $U_R[\{Dom, Eq\}] \leq U$ and, therefore, $\rho(r).Stack$ is of type **Setoid**. It is worth pointing out that, if

$\rho(r).Stack.Dom$ were a record *kind* as studied in (Coquand et al., 2005), we would not be able to introduce a universe to reflect it and hence the above example would not go through (in particular, the refinement map ρ would not be definable). □

4. Dependent Record Types v.s. Σ-types

Dependent record types are arguably better mechanisms than Σ-types when used to represent types of modules. However, some people may still take the view that, although they bring convenience to applications, dependent record types are not necessary — they can always be replaced by Σ-types. In this section, it is argued that such a view is not completely justified. In particular, we consider a case to demonstrate that this is not the case: record types can be used adequately in some situations while Σ-types cannot.

As we know, the only difference between a dependent record type and a Σ-type is that the former has field labels. Our case considers the use of module types together with some form of structural subtyping, in the framework of coercive subtyping (Luo, 1999), and shows that the labels are actually useful in making a finer distinction between record types so that record types can be used adequately in some applications, while Σ-types cannot as they do not have labels.

Module types with structural subtyping. A module type can be represented in a type theory as either a Σ-type or a dependent record type (and, in the non-dependent case, a product type or a non-dependent record type). Here, by structural subtyping for module types, we mean the following subtyping relationships:

- Projective subtyping: a module type is a subtype of its constituent types. For instance, in the framework of coercive subtyping and for the first projection,

$$A \times B \leq_{\pi_1} A \quad \text{and} \quad \langle l_1 : A,\ l_2 : B\rangle \leq_{[\_]} \langle l_1 : A\rangle,$$

  where π_1 and $[\_]$ are the first projection operators for Σ-types and record types, mapping (a, b) to a and $\langle l_1 = a,\ l_2 = b\rangle$ to $\langle l_1 = a\rangle$, respectively.

- Component-wise subtyping: subtyping relationships propagate through the module types. For example, for product types (i.e., Σ-types in the

non-dependent case, and similar for record types — see below), if $A \leq_c A'$ and $B \leq_{c'} B'$, then $A \times B \leq_d A' \times B'$, where d maps (a, b) to $(c(a), c'(b))$ in the component-wise way.

Structural subtyping can be useful for many applications. For example, when using module types to represent classes in an object-oriented language such as Java, it would be desirable for these subtyping relationships to hold between the representing types in order to capture the subclassing relationships between classes. This is elaborated in the following example (see (Luo, 2009) for more details.)

Example 4.1. *A class in an OO-language consists of two parts: states and methods. The former can be represented as a module type and the latter by means of intensional manifest fields as studied in (Luo, 2009). Here, we omit the details of how to represent methods but focus on the representation of states.*

For a class C, its states can be represented as a module type either as $A_1 \times ... \times A_n$ or $\langle l_1 : A_1,\ ...,\ l_n : A_n \rangle$, where A_i's are types. For example, in the type-theoretic model as described in (Luo, 2009), if C is a class, then its type of states is such a module type S_C.

In order to obtain a faithful representation, we would like that the subtyping relationships between the representing types capture the subclassing relationships between classes. Therefore, it would be desirable to have $S_{C'} \leq S_C$ if C' is a subclass of C. This would require that the type of states be a subtype of its constituent types ($S_C \leq A_i$ or $S_C \leq \langle l_i : A_i \rangle$). In the framework of coercive subtyping, this would amount to having both projections from the module types as coercions.

Furthermore, the subtyping relations between the constituent types need to be propagated through the module types and this requires to have component-wise coercions as well. □

Can one consistently make these structural mappings as coercions — are they coherent?[7] Unfortunately, for Σ-types (or product types), one cannot, for otherwise, coherence is lost. It is here that the labels of record types play a crucial role in the coherence of these structural subtyping relations. Particularly, the labels make a special contribution to a more refined dis-

[7]Intuitively, coherence is the condition that the coercions between any two types are unique; that is, a set of coercion rules is coherent if $c = c' : (A)B$ for any coercions c and c' from A to B. See (Luo, 1999) for formal details.

tinction between record types, which is not available for Σ-types. We begin by explaining the coherence problem for Σ-types for structural subtyping.

Incoherence of structural subtyping for Σ-types. It is known from Y. Luo's thesis (Luo, 2005) that, for Σ-types (and product types in the non-dependent case), the following coercions together are incoherent.

- *The first and second projections.* Let's consider the non-dependent case, where the projections are $\pi_1 : (A \times B)A$ and $\pi_2 : (A \times B)B$, for any $A, B : Type$. If we take both projections as coercions, incoherence happens. For instance, taking both A and B to be Nat, π_1 and π_2 are both from $Nat \times Nat$ to Nat, but they are not equal: $\pi_1(3, 5) = 3$ and $\pi_2(3, 5) = 5$.
- *Either projection and the component-wise coercions.* For example, if the first projection and the component-wise mappings were coercions, there would be two different coercions from $(A \times B) \times B$ to $A \times B$: one mapping $((a, b_1), b_2)$ to (a, b_1) (the first projection) and the other mapping $((a, b_1), b_2)$ to (a, b_2) (the composition of the component-wise coercion and the first projection).

Therefore, one cannot use Σ-types (at least in a straightforward way) to represent module types in the applications such as that explained in Example 4.1.

Structural coercions for record types. Although incoherence happens in the above situations for Σ-types and product types, the record types and the corresponding coercions behave in a better way — the labels play a useful role of distinguishing record types from each other. For instance, a record type that corresponds to $Nat \times Nat$ is $Nat_2 \equiv \langle m : Nat,\ n : Nat \rangle$, where the labels m and n are distinct. We may have 'projections' from Nat_2 to $\langle m : Nat \rangle$ and $\langle n : Nat \rangle$, which are two different types – therefore, the record projections are coherent together.

More formally, for non-empty record types,

- the first projection is simply the restriction operation

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

  mapping $\langle r,\ l = a \rangle$ to r, and

- the second projection is the functional operation

$$Snd : (r{:}\langle R,\ l : A \rangle)\langle l : A([r]) \rangle,$$

mapping r to the record $\langle l = r.l \rangle$.

Note that the kind of Snd is different from that of field selection $\_.l$: the codomain type of Snd is the record type $\langle l : A([r]) \rangle$, rather than simply $A([r])$. This makes an important difference: Snd is coherent with the first projection and the component-wise coercions, while field selection is not.

We shall take both of the record projections as coercions. In this paper, only non-dependent coercions (and, in this case, the non-dependent second projection) are studied.[8] Formally, we have the following two coercion rules:[9]

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

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

where, in the second rule above, A is a type, $\langle R,\ l : A \rangle$ stands for $\langle R,\ l : [\_{:}R]A \rangle$, and the kind of Snd is the non-dependent kind $(\langle R,\ l : A \rangle)\langle l : A \rangle$. Note that the label l in the codomain type of Snd is the same label in its domain type.

Remark Label distinction is important. If one allowed label repetitions in record types, as in (Pollack, 2002), the projection coercions $[\_]$ and Snd would be incoherent together. For example, if $Nat_l \equiv \langle l : Nat,\ l : Nat \rangle$ were a well-typed record type, both projections would be from Nat_l to the same type $\langle l : Nat \rangle$, but they are different. □

Component-wise coercions for record types express the idea that coercive subtyping relations propagate through record types: informally, if R is a subtype of R' and A is a 'subtype' of A', then $\langle R,\ l : A \rangle$ is a subtype of $\langle R',\ l : A' \rangle$. Formally, this is formulated by means of the rules in Figure 6.

Remark With nice meta-theoretic properties such as Church-Rosser, we can show that the coercions $[\_]$, Snd and d^i_R $(i = 1, 2, 3)$ are coherent together. Note that, if one used Σ-types instead of record types, we cannot have both projections as coercions (or any projection together with the component-wise coercions) — coherence would have failed, as discussed above. □

[8] When a coercion has a dependent kind, it is a *dependent coercion* (Luo and Soloviev, 1999).

[9] $\Gamma \vdash R \leq_c R' : RType$ is the judgement expressing that the record type R is a subtype of the record type B via coercion c.

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

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

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

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

$$(d^3_R) \qquad \frac{\Gamma \vdash R : RType[L] \quad \Gamma \vdash R' : RType[L] \quad \Gamma \vdash A : (R)Type \quad \Gamma \vdash A' : (R')Type \quad \Gamma \vdash R \leq_c R' : RType \quad \Gamma, x{:}R \vdash A(x) \leq_{c'\{x\}} A'(c(x)) : Type}{\Gamma \vdash \langle R,\ l : A\rangle \leq_{d^3_R} \langle R',\ l : A'\rangle : RType} \quad (l \notin L)$$

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

Figure 6: Component-wise coercions for dependent record types

5. Intensional Manifest Fields in DRTs

As mentioned above, one of the primary functions of dependent record types is for representation of module types. In a type of modules, 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 (see, for example, (MacQueen, 1984; Leroy, 1994) for the use of manifest fields in expressing ML-style sharing).

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 most of the previous studies on manifest fields in type theory (Harper and Lillibridge, 1994; Pollack, 2002; Coquand et al., 2005). 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 (Aspinall, 1995; Hayashi, 1994) and the extensional equality (Martin-Löf, 1984; Constable and Hickey, 2000). 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).

In this section. we briefly demonstrate that manifest fields can be represented for dependent record types with the help of coercive subtyping (Luo, 2009). In particular, we show that the 'sameness' in a manifest field does not have to be interpreted by means of an extensional equality. With the help of coercive subtyping (Luo, 1999), manifest fields are expressible in *intensional* type theories such as Martin-Löf's intensional type theory (Nordström et al., 1990) and UTT (Luo, 1994). 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 is called an *intensional manifest field* (IMF).

5.1. Intensional Manifest Fields in DRTs

An *intensional manifest field* (IMF) in a dependent record type is a field of the form

$$l \sim a : A,$$

where $A : (R)Type$ and $a : (r{:}R)A(r)$. Formally, the above field stands for the following field in a DRT:

$$l : [r{:}R]\mathbf{1}(A(r), a(r)),$$

where $\mathbf{1}(T, t)$ is the unit type parameterised by $T : Type$ and $t : T$ (see §1.1). In other words, we write

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

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

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

The IMFs (and the DRTs 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.

  Formally, 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 d^i_R $(i = 1, 2, 3)$ in Figure 6, that propagate subtyping relations, including those specified by ξ, through DRTs so that the IMFs can be used properly in larger contexts.

Here is a simple example of formalisation of rings in abstract algebra that shows how to use IMFs to represent sharing of domains and illustrates how the coercion ξ can be used to support IMFs.

Example 5.1. *Consider the following module type M, where U is a type universe:*

$$M \equiv \langle S : U,\ op : S \to S,\ ... \rangle.$$

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:

$$M_w \equiv \langle S_w \sim m.S : U,\ op : S_w \to S_w,\ ... \rangle.$$

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

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:

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

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 whose manifest field ensures sharing of the carrier sets:

$$SGw(ag) \quad = \quad \langle B \sim ag.A : U,\ * : B \to B \to B \rangle,$$

where $ag : AG$. Then, the function that generates a ring from an abelian group $ag : AG$ and a semigroup $sg : SGw(ag)$ that share their carrier set can now be defined as:

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

□

Remark The above example shows a particular way that IMFs may be used. In particular, it suggests a way to define the so-called 'with'-clause (see, for example, (Pollack, 2002)) that facilitates sharing by equation: 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 $R_{i-1} \equiv \langle l_1 : A_1, ..., \ l_{i-1} : A_{i-1} \rangle$, we can define the following notation

$$R \ \underline{\texttt{with}} \ l_i \ \underline{\texttt{as}} \ a$$

for the record type

$$\langle \ l_1 : A_1, \ ..., \ l_{i-1} : A_{i-1}, \ \ l_i \sim a : A_i, \ \ l_{i+1} : A_{i+1}, \ ..., \ l_n : A_n \ \rangle.$$

See (Luo, 2009) for the details. □

Experiments in proof assistants.. Experiments on several applications have been done in the proof assistants Plastic (Callaghan and Luo, 2001) and Coq (Coq 2007), both supporting the use of coercions (Luo, 2009). 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[10] 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 5.1 is shown in Figure 7.

6. Meta-theoretic Properties of the Logical Framework with DRTs

Goguen (Goguen, 1994, 1999) has developed a method called *typed operational semantics* (TOS for short) to prove meta-theoretic properties of

[10]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 (Coq 2007) also supports a preliminary form of 'manifest fields' by means of the let-construct, which we do not use in our experiments.

```
(* 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, omitting the 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'').
```

Figure 7: Coq code for the Ring example.

type theories. In his PhD thesis (Goguen, 1994), Goguen has developed the TOS for the type theory UTT and proved that UTT has the nice properties such as Church-Rosser, Subject Reduction and Strong Normalisation.

In this section, using the TOS approach, we study the meta-theoretic properties of the logical framework LF with dependent record types. This is a subsystem, but the basis, of that considered above and we hope that this goes to some lengths to demonstrate the correctness of the DRT formulation, on the one hand, and the usefulness and viability of the TOS approach, on the other.

6.1. The TOS approach: an Informal Introduction

For a type theory, its typed operational semantics captures its computational behaviour, usually given by its (untyped) reduction relation. For example, in TOS, the following judgement

$$\Gamma \models M \to N \to P : A$$

informally asserts that, among other things, N and P are the weak-head normal form and the normal form of the term M, respectively. For the logical framework LF, for example, its corresponding TOS has been studied by Goguen (1999) and its inference rules are given in Figure 8. Since many meta-theoretic properties of a type theory are concerned with its computational behaviour, it is not a surprise that TOS provides an effective approach to the meta-theory of type theories.[11]

The TOS and its corresponding type theory are related to each other by means of the soundness and completeness theorems. Using the judgement $\Gamma \models M : A$ to abbreviate '$\Gamma \models M \to N \to P : A$ for some N and P', we can state the soundness and completeness properties as follows:

- Soundness: $\Gamma \vdash M : A$ implies $\Gamma \models M : A'$ (for A' that is the 'normal form' of A).
- Completeness: $\Gamma \models M : A$ implies $\Gamma \vdash M : A$.

Based on soundness and completeness, we can prove many meta-theoretic properties of the type theory such as strong normalisation.

There are three basic judgement forms in a TOS:

[11] It is worth noting that, although it is useful to study the meta-theory for many type theories, the TOS approach would not be suitable for non-normalising type theories. See (Goguen, 1994) for discussions.

Contexts

$$\frac{}{\models () \to ()} \qquad \frac{\models \Gamma \to \Delta \;\; \Gamma \models A \to B \;\; x \notin FV(\Gamma)}{\models \Gamma, x{:}A \to \Delta, x{:}B}$$

Kinds

$$\frac{\Gamma \models ok}{\Gamma \models Type \to Type} \qquad \frac{\Gamma \models M \to N \to P : Type}{\Gamma \models El(M) \to El(P)}$$

$$\frac{\Gamma \models A_1 \to B_1 \;\; \Gamma, x{:}A_1 \models A_2 \to B_2}{\Gamma \models (x{:}A_1)A_2 \to (x{:}B_1).B_2}$$

Terms

$$\frac{\Gamma_0, x{:}A, \Gamma_1 \models A \to B}{\Gamma_0, x{:}A, \Gamma_1 \models x \to x \to x : B}$$

$$\frac{\Gamma \models A_1 \to B_1 \;\; \Gamma, x{:}A_1 \models M_0 \to_n P_0 : B_2 \;\; [x{:}B_1]P_0 \; not \; \eta{-}redex}{\Gamma \models [x{:}A_1]M_0 \to [x{:}A_1]M_0 \to [x{:}B_1]P_0 : (x{:}B_1)B_2}$$

$$\frac{\Gamma \models A_1 \to B_1 \;\; \Gamma, x{:}A_1 \models M_0 \to_n P(x) : B_2 \;\; \Gamma \models P \to P \to P : (x{:}B_1)B_2}{\Gamma \models [x{:}A_1]M_0 \to [x{:}A_1]M_0 \to P : (x{:}B_1)B_2}$$

$$\frac{\begin{array}{c}\Gamma \models M_1 \to N_1 \to P_1 : (x{:}B_1)B_2 \;\; \Gamma \models M_2 \to N_2 \to P_2 : B_1 \\ \Gamma \models [M_2/x]B_2 \to C \;\; N_1 \; not \; abstraction\end{array}}{\Gamma \models M_1(M_2) \to N_1(M_2) \to P_1(P_2) : C}$$

$$\frac{\begin{array}{c}\Gamma \models M_1 \to_w [x{:}A_1]N_0 : (x{:}B_1)B_2 \;\; \Gamma \models M_2 : B_1 \\ \Gamma \models [M_2/x]N_0 \to P \to Q : C \;\; \Gamma \models [M_2/x]B_2 \to C\end{array}}{\Gamma \models M_1(M_2) \to P \to Q : C}$$

Figure 8: TOS rules for LF.

- $\models \Gamma \to \Delta$: the context Γ has context Δ as its normal form;
- $\Gamma \models A \to B$: the kind A is well-formed in context Γ and has normal form B; and
- $\Gamma \models M \to N \to P : A$: the terms M, N, P are well-formed in context Γ of kind A and M has weak-head normal form N and normal form P.

From these basic judgements, one can define other forms of judgements, including the following:

- $\Gamma \models$ *ok* stands for '$\models \Gamma \to \Delta$ for some Δ';
- $\Gamma \models M \to_w N : A$ stands for '$\Gamma \models M \to N \to P : A$ for some P';
- $\Gamma \models M \to_n P : A$ stands for '$\Gamma \models M \to N \to P : A$ for some N'; and
- $\Gamma \models M : A$ stands for '$\Gamma \models M \to N \to P : A$ for some N and P'.

6.2. Typed Operational Semantics for LF with DRTs

The typed operational semantics for the logical framework LF with DRTs is the extension of the TOS for LF (Figure 8) with the inference rules given in Figure 9, most of which are self-explanatory. We only mention that, besides using the abbreviated forms of judgement (see above) in the rules, we also use the terminology of 'pair-records', which refer to the terms of the form $\langle r,\ l = a : A\rangle$. For example, in $(BASE_{RESTR})$, we require that p or q be not a pair-record, for otherwise, for instance, $[p]$ could be a redex and would not be in normal form.

The TOS has many important properties including, for example, the following lemma of derterminacy and strong normalisation theorem.

Lemma 6.1 (determinacy).

- *If* $\models \Gamma \to \Delta$ *and* $\models \Gamma \to \Phi$, *then* $\Delta \equiv \Phi$.
- *If* $\Gamma \models A \to B$ *and* $\Gamma \models A \to C$, *then* $B \equiv C$.
- *If* $\Gamma \models M \to N \to P : B$ *and* $\Gamma \models M \to Q \to R : C$, *then* $N \equiv Q$, $P \equiv R$ *and* $B \equiv C$.

Theorem 6.2 (Strong Normalisation of TOS).

1. *If* $\Gamma \models A \to B$ *then* A *is strongly normalisable.*

Kinds of Record Types

$$\frac{\Gamma \models ok}{\Gamma \models RType \to RType}\ RTYPE \qquad \frac{\Gamma \models ok}{\Gamma \models RType[L] \to RType[L]}\ RTYPE[L]$$

Record Types

$$\frac{\Gamma \models ok}{\Gamma \models \langle\rangle \to \langle\rangle \to \langle\rangle : RType[\emptyset]}\ EMP_{RCDT}$$

$$\frac{\Gamma \models R \to_n P : RType[L] \quad \Gamma \models A \to_n B : (P)Type \quad l \notin L}{\Gamma \models \langle R, l : A\rangle \to \langle R, l : A\rangle \to \langle P, l : B\rangle : RType[L \cup \{l\}]}\ RCDT$$

Pair-records

$$\frac{\Gamma \models ok}{\Gamma \models \langle\rangle \to \langle\rangle \to \langle\rangle : \langle\rangle}\ EMP_{RCD}$$

$$\frac{\begin{array}{c}\Gamma \models \langle R, l : A\rangle \to_n \langle P, l : B\rangle : RType \quad \Gamma \models r \to_n p : P \\ \Gamma \models A(r) \to_n C : Type \quad \Gamma \models a \to_n b : C\end{array}}{\Gamma \models \langle r, l = a : A\rangle \to \langle r, l = a : A\rangle \to \langle p, l = b : B\rangle : \langle P, l : B\rangle}\ RCD$$

Restrictions

$$\frac{\Gamma \models r \to q \to p : \langle P, l : B\rangle \quad p,\ q\ not\ pair\text{-}records}{\Gamma \models [r] \to [q] \to [p] : P}\ BASE_{RESTR}$$

$$\frac{\Gamma \models r \to_w \langle p, l = b : A\rangle : \langle P, l : B\rangle \quad \Gamma \models p \to s \to t : P}{\Gamma \models [r] \to s \to t : P}\ RESTR$$

Selections

$$\frac{\begin{array}{c}\Gamma \models r \to q \to p : \langle P, l : B\rangle \quad p,\ q\ not\ pair\text{-}records \\ \Gamma \models B([r]) \to_n C : Type\end{array}}{\Gamma \models r.l \to q.l \to p.l : C}\ BASE_{FLDSEL}$$

$$\frac{\Gamma \models r \to_w \langle p,\ l = b : A\rangle : \langle P,\ l : B\rangle \quad \Gamma \models b \to c \to d : C \quad \Gamma \models A(p) \to_n C : Type}{\Gamma \models r.l \to c \to d : C}\ FLDSEL$$

$$\frac{\Gamma \models r \to_n s : \langle P, l : B\rangle \quad \Gamma \models [r].l' \to c \to d : C \quad l \neq l'}{\Gamma \models r.l' \to c \to d : C}\ FLDSL'$$

Figure 9: Inference Rules of Typed Operational Semantics for LF with DRTs

2. *If* $\Gamma \models M \to N \to P : A$ *then* M *is strongly normalisable.*

For the TOS, we can prove the soundness and completeness theorems.

Theorem 6.3 (Completeness).

- *If* $\Gamma \models ok$ *then* $\vdash \Gamma$ *valid.*
- *If* $\Gamma \models A \to B$ *then* $\Gamma \vdash A$ *kind and* $\Gamma \vdash A = B$.
- *If* $\Gamma \models M \to N \to P : A$ *then* $\Gamma \vdash M : A$, $\Gamma \vdash M = N : A$, $\Gamma \vdash M = P : A$ *and* $\Gamma \vdash A = A$.

Theorem 6.4 (Soundness).

- *If* $\Gamma \vdash ok$, *then there exists* Δ *such that* $\models \Gamma \to \Delta$.
- *If* $\Gamma \vdash A$ *kind, then there exists* B *such that* $\Gamma \models A \to B$.
- *If* $\Gamma \vdash A = B$ *then there exists* C *such that* $\Gamma \models A \to C$ *and* $\Gamma \models B \to C$.
- *If* $\Gamma \vdash M : A$ *then there exist* P, B *such that* $\Gamma \models A \to B$ *and* $\Gamma \models M \to_n P : B$.
- *If* $\Gamma \vdash M = N : A$, *then there exist* P, B *such that* $\Gamma \models A \to B$, *and* $\Gamma \models M \to_n P : B$, $\Gamma \models N \to_n P : B$.

Proof. By induction on derivations. For the cases of LF-rules, see (Goguen, 1999). We consider the following two cases about record types.

- The second introduction rule in Figure 1:

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

  By induction hypothesis, the following hold:

  1. $\Gamma \models \langle R, l : A\rangle \to_n \langle P, l : B\rangle : RType$ for some P and B,
  2. $\Gamma \models r \to_n p : P'$ and $\Gamma \models R \to_n P' : RType[L]$ for some p, P' and L, and
  3. $\Gamma \models a \to_n b : C$ and $\Gamma \models A(r) \to C$ for some b and C.

  By Determinacy Lemma 6.1 and inversion of the rule $(RCDT)$ in Figure 9, $P \equiv P'$. Therefore, by rule (RCD) in Figure 9, $\Gamma \models \langle r, l = a : A\rangle \to_n \langle p, l = b : B\rangle : \langle P, l : B\rangle$.

- The third elimination rule in Figure 1:

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

By induction hypothesis, the following hold:

1. $\Gamma \models r \to_n s : \langle P, l : B\rangle$ and $\Gamma \models \langle R, l : A\rangle \to \langle P, l : B\rangle$ for some s, P and B, and
2. $\Gamma \models [r].l' \to_n c : C$ and $\Gamma \models B \to C$ for some c and C.

Since $l \neq l'$, by $(FLDSL')$ in Figure 9, we have $\Gamma \models r.l' \to_n c : C$.

6.3. Meta-theoretic properties of DRTs

From the properties of the TOS, we can prove the following meta-theoretic properties of LF with DRTs (for their proofs, see (Feng and Luo, 2010)).

Theorem 6.5. *For LF with DRTs, we have*

- *Church-Rosser: If* $\Gamma \vdash M = N : A$, *then* $M \to^* P$ *and* $N \to^* P$ *for some* P.
- *Subject Reduction: If* $\Gamma \vdash M : A$ *and* $M \to N$, *then* $\Gamma \vdash N : A$.
- *Strong Normalisation*
  1. *If* Γ *valid, then* Γ *is strongly normalisable.*
  2. *If* $\Gamma \vdash A$ *kind, then* A *is strongly normalisable.*
  3. *If* $\Gamma \vdash A = B$, *then both* A *and* B *are strongly normalisable to some* C.
  4. *If* $\Gamma \vdash M : A$, *then* M *and* A *are strongly normalisable and* $\Gamma \vdash P : B$, *where* P *and* B *are the normal forms of* M *and* A, *respectively.*
  5. *If* $\Gamma \vdash M = N : A$, *then both* M *and* N *are strongly normalisable to some* P, A *is strongly normalisable to some* B *such that* $\Gamma \vdash P : B$.

7. Related Work and Conclusion

In this paper, we have studied dependent record types, their applications and some of the meta-theoretic properties. Some of the related and future work is discussed here.

Related work. As mentioned in the introduction, there have been several studies of dependently-typed records, many of which study dependent record kinds at the level of the logical framework. In particular, Coquand et al. (2005) have studied the meta-theoretic properties of dependent record kinds, including the decidability of type-checking, among others. As discussed in §2.1, kinds have a simpler structure and that allows one to conduct meta-theoretic studies in a more straightforward way. For instance, as a simple example, it is straightforward to determine the set of top-level labels of a record kind since it is always of the form $\langle l_1 : A_1,\ ...,\ l_n : A_n \rangle$, while this is not the case for record types. This is why the meta-theoretic studies of record types are more sophisticated as in §6 and (Feng and Luo, 2010).

Pollack (2002) studies dependent record types and shows very interesting examples of how dependently-typed records may be used in formalisation of mathematics. However, the paper did not show that record types are stronger and more useful than record kinds – the examples in that paper can also be done with record kinds. The current paper has studied DRTs further in this respect and compared them with Σ-types.

In the formulation of DRTs in the present paper, we have introduced the kinds $RType[L]$ of the record types, where the associated label set L plays a crucial role in forming record types with distinct labels. This is one of the important improvements in formulation of DRTs for dependent type theories. In (Pollack, 2002), labels may be repeatedly used in a record type, although a latter label overrides a former one. This was considered because, unlike record kinds, a record type can take other syntactic forms other than $\langle l_1 : A_1,\ ...,\ l_n : A_n \rangle$ and, therefore, it has been a problem how to determine its label set. Our kinds $RType[L]$ play this role to make it possible to form DRTs with only distinct labels.

Dependently typed records have been implemented in several proof assistants. In Coq (Coq 2007), dependent record types are actually implemented as inductive types (or generalised Σ-types) with labels defined as global names. As a consequence, for example, different record types must have distinct labels in the same environment. Coq provides useful mechanisms with its records implementation. For example, it provides a very useful mechanism that allows one to specify easily projections as coercions. The Agda proof assistant (Agda 2008) implements a notion of record type that is more flexible and stronger, with even the so-called recursive records allowed. This kind of experimental features certainly require careful research into them to obtain a better understanding.

Future work. The DRTs studied in this paper are intensional. This is different from that presented in (Luo, 2009), where 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:

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

For example, for any $r : \langle R,\ l : A\rangle$ (r can be a variable), we have, by the second rule above, that $r = \langle [r],\ l = r.l : A\rangle : \langle R,\ l : A\rangle$. It is unclear whether such extensionality is useful in practice. Further studies are called for to investigate this.

We have studied the relationships between DRTs and Σ-types. This may have raised concerns, as well as interests, between their formal relations. Formally, we'd like to prove that adding DRTs does not increase the logical power of the system; in other words, adding DRTs is a conservative extension of a type theory with Σ-types (and other constructions) already present in the type theories such as Martin-Löf's type theory or UTT. Adams and the author have made efforts in conducting this research, which have reduced the problem of conservativity to that of injectivity of record types[12] (Admas and Luo, 2011). Injectivity is not a trivial property and one of the ways to prove it is to use Church-Rosser of the system with DRTs, which may be proved by extending the TOS approach as described in §6 to more sophisticated type theories. Research efforts are needed to follow this line of possibly tedious but useful research.

References

Admas, R., Luo, Z., 2011. Dependent record types and Σ-types compared. Manuscript.

Agda 2008, 2008. The Agda proof assistant (version 2). http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php.

Aspinall, D., 1995. Subtyping with singleton types, in: CSL'94, LNCS'993.

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

[12]Formally, injectivity of record types is the property that, if $\Gamma \vdash \langle R,\ l : A\rangle = \langle R',\ l' : A'\rangle : RType$, then $\Gamma \vdash R = R' : RType$ and $\Gamma \vdash A = A' : Type$.

Betarte, G., Tasistro, A., 1998. 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.

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

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

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

Coquand, T., Huet, G., 1988. The calculus of constructions. Information and Computation 76.

Coquand, T., Pollack, R., Takeyama, M., 2005. A logical framework with dependently typed records. Fundamenta Informaticae 65.

Feng, Y., Luo, Z., 2010. Typed operational semantics for dependent record types, in: Proceedings of Types for Proofs and Programs (TYPES'09), Aussois, France. EPTCS 53.

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

Goguen, H., 1999. Soundness of typed operational semantics for the logical framework. Typed Lambda Calculi and Applications (TLCA'99), LNCS'1581 .

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

Hayashi, S., 1994. Singleton, union and intersection types for program extraction. Information and Computation 109, 174–210.

Leroy, X., 1994. Manifest types, modules and separate compilation. POPL 1994 .

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

Luo, Z., 1993. Program specification and data refinement in type theory. Mathematical Structures in Computer Science 3.

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

Luo, Z., 1997. Coercive subtyping in type theory. CSL'96, LNCS'1258 .

Luo, Z., 1999. Coercive subtyping. J of Logic and Computation 9, 105–130.

Luo, Z., 2009. Manifest fields and module mechanisms in intensional type theory, in: Types for Proofs and Programs, Proc. of Inter. Conf. of TYPES'08. LNCS 5497.

Luo, Z., 2010. Dependent record types revisited, in: Proc. of the 1st Inter. Workshop on Modules and Libraries for Proof Assistants (MLPA'09), Montreal. ACM Inter. Conf. Proceeding Series 429.

Luo, Z., Luo, Y., 2005. Transitivity in coercive subtyping. Information and Computation 197, 122–144.

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

Luo, Z., Soloviev, S., 1999. Dependent coercions. Proc of the 8th Inter. Conf. on Category Theory in Computer Science (CTCS'99), Electronic Notes in Theoretical Computer Science, Vol 29. .

MacQueen, D., 1984. Modules for standard ML. ACM Symp. on Lisp and Functional Programming .

Martin-Löf, P., 1984. Intuitionistic Type Theory. Bibliopolis.

Matita 2008, 2008. The Matita proof assistant. http://matita.cs.unibo.it/.

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

Pollack, R., 2002. Dependently typed records in type theory. Formal Aspects of Computing 13, 386–402.

Saïbi, A., 1997. Typing algorithm in type theory with inheritance. POPL 1997 .

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