Dependent Record Types Revisited

Zhaohui Luo
Department of Computer Science
Royal Holloway, University of London
Egham, Surrey TW20 0EX, U.K.
zhaohui@cs.rhul.ac.uk

ABSTRACT

Dependently-typed records have been studied in type theory in several previous research attempts, with applications to the study of module mechanisms for both programming and proof languages. Recently, the author has proposed an improved formulation of dependent record types in the context of studying manifest fields of module types. In this paper, we study this formulation in more details by considering universes of record types and some application examples. In particular, we show that record types provide a more powerful mechanism (than record kinds) in expressing module types and additional useful means (as compared with Σ-types) in applications.

1. INTRODUCTION

Dependently-typed records have been studied in type theory in several previous research attempts [7, 1, 16, 3], with applications to the study of module mechanisms for both programming and proof languages. Recently, the author has proposed an improved formulation of dependent record types in the context of studying manifest fields of module types [12]. In this paper, we study this formulation in more details by considering universes of record types and some application examples.

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 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,[1] with [16] as the only 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, 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 [16], 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 also 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, the 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.

The following subsection briefly describes the logical framework LF, establishing notational conventions. Dependent record types are formulated in §2, where we also discuss several issues concerning the design decisions. In §3, we show how to introduce universes of dependent record types and illustrate 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.

1.1 The Logical Framework LF

LF [10] is the typed version of Martin-Löf's logical framework [15]. It is a dependent type system for specifying type theories. The types in LF are called *kinds*, including:

- $Type$ — the kind representing the collection of all types (A is a type if $A : Type$);
- $El(A)$ — the kind of objects of type A (we often omit El); and

[1]For example, both [1] and [3] study record kinds – their 'record types' are studied at the level of kinds in a logical framework.

- $(x{:}K)K'$ (or simply $(K)K'$ when $x \notin FV(K')$) — the kind of dependent functional operations f which can be applied to an object k of kind K to form $f(k)$ of kind $[k/x]K'$. When $f \equiv [x{:}K]k$, $f(a)$ is computationally equal to $[a/x]k$.

We use $\equiv$ to denote the syntactical identity (up to α-conversion) and write $M\{x\}$ to indicate that x may occur free in M and subsequently write $M\{a\}$ for the substitution $[a/x]M$.[2]

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 $\lambda(A, B, f)$ that, when applied to a of type A, returns an object that is equal to $f(a)$ of type $B(a)$, and
  - Σ-types $\Sigma(A, B)$ of dependent pairs (a, b)[3] with π_1 and π_2 being the associated projection operators.

  $A \to B$ and $A \times B$ will be used for non-dependent Π-type and Σ-type, respectively.

Type theories thus specified in LF are intensional type theories such as Martin-Löf's intensional type theory [15] and the Unifying Theory of dependent Types (UTT) [10]. Intensional type theories have nice meta-theoretic properties including Church-Rosser, Subject Reduction and Strong Normalisation (see Goguen's thesis on the meta-theory of UTT [4, 5]).

2. DEPENDENT RECORD TYPES

In this section, we present the formulation of dependent record types, as given in [12], followed by some discussions on its design decisions.

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:

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

[2]Usually, we would have instead used $M[\_]$ for $M\{\_\}$. In this paper, however, $M\{\_\}$ is used to differentiate it from the restriction operator $[\_]$ for records.

[3]We use the notation of untyped pairs — see, for example, [12] for an explanation of how this is possible, thanks to coercive subtyping.

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:

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

These kinds obey obvious subkinding relationships:

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

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, indexed by the objects of R, and this is how dependency is embodied in the formulation.

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 [1]):

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

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 the associated operations, 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 $n \neq n'$.

Notation We shall adopt the following notational conventions.

- 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

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

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

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

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.

- 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 (see Appendix A of [12] for an explanation). □

2.2 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 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 [1] and [3] study record *kinds* — their 'record types' are studied at the level of kinds in the logical framework, while only [16] studies record types.)

Record types are much more powerful than record kinds. As explained later in §3, 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., [3]), 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.[4] It is probably because of this difficulty that, when record types are studied in [16], 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 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 $Extend[l](R) =_{\text{df}} \langle R,\ l : [x : R]Nat\rangle$, 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 [12] 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 [7, 1, 3]. In this respect, [16] 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

[4] There is a problem in [1], 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.

adopt more flexible subtyping relations in formalisation and modelling.

3. UNIVERSES OF RECORD TYPES AND APPLICATIONS

Universes of dependent record types and their use in applications are considered in this section. We explain in §3.1 how universes of record types can be introduced and then illustrate their use in §3.2 by giving an example in data refinement which, in particular, shows that record types provide a more powerful mechanism than record kinds.

3.1 Universes of Dependent Record Types

One may collect (the names of) some types into a type called a universe [14]. 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 [14] or §9.2.3 of [10]). 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.[5] We introduce the following universes:

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

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

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

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

Names of the record types are introduced into the universes $U_R[L]$ by the rules in Figure 2. 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 3.

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. □

3.2 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 §3.1, they provide a more powerful mechanism for module types than record kinds, as the example below in data refinement illustrates.

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

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

- 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 [9]. 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 [9]; 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 §3) 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., $Stack.Eq(pop(push(n, s)), s)$).*

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

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

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

Figure 2: 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 3: Subtyping between universes.

- **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:*

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

  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*

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

  as follows: for any record $r : \textbf{Str}[\textbf{Array}(Nat)]$*,* $\rho(r)$ *is defined to be the record given in Figure 4. In the implementation, a stack is represented 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 [3], 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. $\sum$-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 [11], 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.

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

Figure 4: Refinement map from arrays to stacks.

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 [12] 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 [12]. 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 [12], 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?[6] 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 distinction 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 [8] 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

[6] 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 [11] for formal details.

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

Figure 5: Component-wise cubtyping for record types.

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.[7] Formally, we have the following two coercion rules:[8]

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

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

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.

[7]When a coercion has a dependent kind, it is a *dependent coercion* [13].

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

Remark Label distinction is important. If one allowed label repetitions in record types, as in [16], 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 rule in Figure 5: where, for any $r_0 : \langle R,\ l : A\rangle$,

$$d_R(r_0) =_{\mathrm{df}} \langle c([r_0]),\ l = c'\{[r_0]\}(r_0.l)\rangle,$$

mapping $\langle r,\ l = a\rangle$ to $\langle c(r),\ l = c'\{r\}(a)\rangle$.

Remark Assuming that the extension with dependent record types has nice meta-theoretic properties such as Church-Rosser, we can show that the coercions $[\_]$, Snd and d_R 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. □

5. CONCLUSION

In this paper, dependent record types are studied with respect to their formulation and applications. As to future work, we would like to see the development of the meta-theory of dependent record types (e.g., along the line of Typed Operational Semantics [4, 6]) and a proper implementation of dependent record types in proof assistants[9] so that they can be effectively used in future applications.

Acknowledgement This work is partially supported by the research grant F/07-537/AA of the Leverhulme Trust in U.K.

6. REFERENCES

[1] G. Betarte and A. Tasistro. Extension of Martin-Löf's type theory with record types and subtyping. In G. Sambin and J. Smith, editors, *Twenty-five Years of Constructive Type Theory*. Oxford University Press, 1998.

[9]Coq [2], for example, only supports a macro for dependent record types, but not proper record types. It is a macro in the sense that dependent record types are actually implemented as inductive types (a general form of Σ-types) with labels as defined global terms (in other words, they are not labels in the proper sense). There are undesirable consequences of this; for instance, the 'labels' of different 'record types' must be different.

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

[3] T. Coquand, R. Pollack, and M. Takeyama. A logical framework with dependently typed records. *Fundamenta Informaticae*, 65(1-2), 2005.

[4] H. Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, University of Edinburgh, 1994.

[5] H. Goguen. The metatheory of UTT. In *Types for Proofs and Programs, Proc. of Inter. Conf. of TYPES'94. LNCS 996*, 1995.

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

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

[8] Y. Luo. *Coherence and Transitivity in Coercive Subtyping*. PhD thesis, University of Durham, 2005.

[9] Z. Luo. Program specification and data refinement in type theory. *Mathematical Structures in Computer Science*, 3(3), 1993.

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

[11] Z. Luo. Coercive subtyping. *J of Logic and Computation*, 9(1):105–130, 1999.

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

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

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

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

[16] R. Pollack. Dependently typed records in type theory. *Formal Aspects of Computing*, 13:386–402, 2002.