Dependently Typed Records for Representing Mathematical Structure[*†]

Robert Pollack
Durham University
rap@dcs.ed.ac.uk

June 1, 2000

1 Introduction

Consider a statement about groups "*For G a group, ...*". A naive approach to formalize this is to unfold the meaning of group, so that every statement about groups begins with

$$\textit{For } G \textit{ a set, } + \textit{ an operation on } G, + \textit{ associative, } e \in G, \ldots \tag{1}$$

This "unpackaged" approach can be improved by collecting all the parts of the meaning of group into a *context*, which need not be explicitly mentioned in every statement. A means of *discharging* some of the context is provided, so that statements made under that context can be instantiated with particular groups. However once the group context is discharged, all the parts of a group must be mentioned when using any general lemma about groups. Variations on this are supported by many proof tools, e.g. *Coq*'s `Section` mechanism [5], *Lego*'s `Discharge` [12], *Automath* contexts and *Isabelle* locales.

A significant refinement is achieved by giving names to bits of context as in *telescopes* [7], or first-class contexts as in Martin-Löf's framework with explicit substitutions [19]. With these, we need not discharge a context to instantiate definitions and lemmas. But contexts or telescopes are "flat"; they don't show that structures are built from existing structures, sharing some parts, and inheriting some properties.

We informally define "packaging" as any approach to collecting the parts of mathematical structures, supporting more abstract manipulation of structures. Packaging is a two-edged sword: once structures are packaged to gain abstraction, we need more tools for manipulating them.

1.1 Overview

In Sect. 2 we show well-known inductively definable packaging constructions, Sigma types and inductive records, pointing out some issues of efficiency and abstraction that are not so well-known. Section 3 presents true records, including treatment of labels. This presentation

*Supported by UK EPSRC grant GR/M75518

†A previous version of this paper appears in *Theorem Proving in Higher Order Logics, TPHOLs 2000*, See `www.springer.de/comp/lncs/`.

is inspired by the work of Betarte and Tasistro [4, 3, 19], but considerably simplifies and explains that work. One of our simplifications is removal of record subtyping from the core description of records, in favor of the more general notion of *coercive subtyping*, which is briefly discussed in Sect. 4.

Sect. 5, about making signatures more precise, discusses *Pebble style sharing* and *manifest types* in signatures. The manifest types are extended by a simple *with* notation for adding manifest information to an existing signature. Sect. 6 shows manifestly typed records as a definitional extension of type theory with inductive-recursive definition.

The main novelty of the present paper is a simple, but completely formal treatment of manifest types and *with* in signatures. (My formal presentation is *unitary* in the sense that "..." appears only in informal discussion.) The main idea I wish to sell is that manifest signatures are necessary for the actual (in practice) formalization of mathematics.

Some notations I use $[x{:}\,A]M$ for lambda-abstraction, $(x{:}\,A)B$ for dependent function type, $(A)B$ for non-dependent function type and $M(N)$ for application. $[\_{:}\,A]M$ is a lambda-abstraction whose variable is not used in the body. Field labels are written $\mathtt{r}$, $\mathtt{p}$, to distinguish them from variables r, p.

Acknowledgement This paper owes much to discussion with my colleagues in the Computer Assisted Reasoning Group at Durham, especially Zhaohui Luo.

2 Definable Structures

We begin with some structuring techniques that are inductively definable in dependent type theory. These are first-class constructions, and as such can be parameterized using lambda-abstraction.

I will use partial equivalence relation (PER) as a running example. Let sym (resp. trn) express that R is a symmetric (resp. transitive) relation over S. PER is the telescope (2) or the informal record type (3)

$$[S{:}\,\mathtt{Set}][R{:}\,(S)(S)\mathtt{Prop}][sAx{:}\,sym(S)(R)][tAx{:}\,trn(S)(R)], \tag{2}$$

$$\langle S{:}\,\mathtt{Set},\ R{:}\,(S)(S)\mathtt{Prop},\ sAx{:}\,sym(S)(R),\ tAx{:}\,trn(S)(R)\rangle. \tag{3}$$

An object with the signature (or *fitting in to* the telescope) PER is informally

$$\langle S{=}T,\ R{=}Q,\ sAx{=}\,symQ,\ tAx{=}\,trnQ\rangle.$$

where $T{:}\,\mathtt{Set}$, $Q{:}\,(T)(T)\mathtt{Prop}$, $symQ{:}\,sym(T)(Q)$ and $trnQ{:}\,trn(T)(Q)$.

2.1 Sigma Types

It is clear from (1) that any approach to packaging must handle dependency of later parts on earlier parts of the package. The simplest dependent package is *pairs* with *Sigma types*. For our purposes it is best to use a *logical framework* presentation of Sigma types (Fig. 1), where dependency is handled by the dependent function type of the framework (see rule Form). While I have written the computation rules as typed equality, they can be implemented by syntactic reduction.

$$\textsc{Form}\ \frac{A : \mathtt{Type} \qquad P : (A)\mathtt{Type}}{\Sigma AP : \mathtt{Type}} \qquad \textsc{Intro}\ \frac{\Sigma AP : \mathit{Type} \qquad a : A \qquad p : P(a)}{\langle a, p\rangle_{\Sigma AP} : \Sigma AP}$$

$$\textsc{Elim1}\ \frac{p : \Sigma AP}{p.1 : A} \qquad \textsc{Elim2}\ \frac{p : \Sigma AP}{p.2 : P(p.1)}$$

$$\textsc{Comp1}\ \frac{\langle a, p\rangle_{\Sigma AP} : \Sigma AP}{\langle a, p\rangle_{\Sigma AP}.1 = a : A} \qquad \textsc{Comp2}\ \frac{\langle a, p\rangle_{\Sigma AP} : \Sigma AP}{\langle a, p\rangle_{\Sigma AP}.2 = p : P(a)}$$

Figure 1: LF presentation of Sigma types.

For type synthesis to be effective, the pairs, $\langle a, p\rangle_{\Sigma AP}$, must be *heavily typed*, i.e. carry the annotation ΣAP.[1] But then the type of the first component can be inferred, so I will write ΣP and $\langle a, p\rangle_{\Sigma P}$.

Sigma can be formalized in *Coq* (and similarly in *Lego*) as an inductively defined family with a single constructor:

```
Inductive sigT [A:Type; P:A->Type]: Type :=
              existT: (x:A)(P x) -> (sigT A P).
```

The constructor of `(sigT A P)` is `(existT A P)`; i.e. the pairs are heavily typed, as in Fig. 1. The two projections are defined in terms of the inductive elimination rule (i.e. by case analysis); e.g. the first projection

```
projT1 [A:Type; P:A->Type; H:(sigT A P)]: A :=
                      Cases H of (existT x _) => x end.
```

Unlike Fig. 1, definable projection is heavily typed, but that is an artifact of this functional presentation.

2.1.1 Example: right association

We can represent *PER* by associating pairs to the right. This is usually written out directly

$$\mathit{PER} := \Sigma[S\colon \mathtt{Set}]\Sigma[R\colon (S)(S)\mathtt{Prop}]\Sigma[\_\colon \mathit{sym}(S)(R)]\mathit{trn}(S)(R). \tag{4}$$

To clarify, this definition can be written incrementally

$$\begin{array}{rlcl} \mathit{Inner} & [S\colon \mathtt{Set}][R\colon (S)(S)\mathtt{Prop}] & := & \Sigma[\_\colon \mathit{sym}(S)(R)]\mathit{trn}(S)(R), \\ \mathit{Middle} & [S\colon \mathtt{Set}] & := & \Sigma[R\colon (S)(S)\mathtt{Prop}]\mathit{Inner}(S)(R), \\ \mathit{PER} & & := & \Sigma[S\colon \mathtt{Set}]\mathit{Middle}(S). \end{array} \tag{5}$$

Inside the signature, the fields are named and referred to by bound variables, S and R. However, these names are purely local; to refer to the fields of a *PER* object we must use the anonymous first and second projections. To help matters we can give names to field

[1] For example, an un-annotated pair, $\langle a, p\rangle$, inhabits both ΣAP and $A \times P(a)$, which are not equal types.

projectors. (I use informal *dot notation* for application of these defined projectors, although they are actually functions.)

$$\begin{array}{rlcll} S & [P\colon PER] & : & \mathtt{Set} & := P.1 \\ R & [P\colon PER] & : & (P.S)(P.S)\mathtt{Prop} & := P.2.1 \\ sAx & [P\colon PER] & : & sym(P.S)(P.R) & := P.2.2.1 \\ tAx & [P\colon PER] & : & trn(P.S)(P.R) & := P.2.2.2 \end{array}$$

These defined field projectors are global, and cannot be reused globally (e.g. as projectors in other packages) without shadowing or another ad hoc solution.

Here is an inhabitant of the signature defined in (4)

$$\langle T, \langle Q, \langle symQ, trnQ\rangle_{Inner(T)(Q)}\rangle_{Middle(T)}\rangle_{PER}. \tag{6}$$

2.1.2 Example: left association

An alternative is to associate pairs to the left.

$$\begin{array}{rcl} Rel & := & \Sigma[S\colon \mathtt{Set}](S)(S)\mathtt{Prop}, \\ symRel & := & \Sigma[P\colon Rel]sym(P.1)(P.2), \\ PER & := & \Sigma[P\colon symRel]trn(P.1.1)(P.1.2). \end{array} \tag{7}$$

We do not get to use the local field names directly when defining this signature, but have to project the fields, e.g. $P.1.1$. As above, we can define top level names for field projectors of this tuple:

$$\begin{array}{rlcll} S & [P\colon PER] & : & \mathtt{Set} & := P.1.1.1 \\ \vdots & & & & \\ tAx & [P\colon PER] & : & trn(P.S)(P.R) & := P.2 \end{array}$$

Here is an inhabitant of the signature defined in (7)

$$\langle\langle\langle T, Q\rangle_{Rel}, symQ\rangle_{symRel}, trnQ\rangle_{PER}. \tag{8}$$

2.1.3 Some Differences

By giving both right and left associating definitions of *PER* incrementally I have emphasised the duality of these constructions. However this "symmetry" is broken because type dependency increases to the right. In Sect. 5.2.2 I argue that left association is really better; in the meantime, here are three differences.

The nested pairs of (6) and (8) are heavily typed, and there is duplication of type information in both cases, so an implementation may optimize structures by keeping only the outermost type annotation.[2] The inner annotations of (8) are subterms of outer annotations, so projection from left associating structures is cost-free using this optimization. However in (6), the inner annotations are substitution instances of outer annotations, so this optimization requires traversing types in order to project the second component.

If we want to add another field to a structure, (e.g. to make *PER* into *equivalence relation*), it must be added on the right, as it will, in general, depend on all the previous fields. Thus (7) can be extended directly and the structure will remain left associating and have *PER* as an

[2] *Lego* does this with built-in Sigma types.

immediate substructure. This property is called *extensibility.* On the other hand, extending (4) will either entail breaking the right-association or reorganizing the entire structure so that *PER* is no longer a substructure.

Suppose we want to specialize *PER* to the natural numbers. The data of definition (5) shows directly how the rest of the package depends on the first field; e.g. $Middle(nat)$ is the structure we want. Such "application" of parameterized signatures is known as *Pebble style sharing*, and will be discussed further in Sect. 5.1. There is no obvious way to specialize definition (7) without reorganizing the entire structure. This point is evidently dual to the previous paragraph. The observation that "application" for sharing can be defined directly on nested Sigma types appears in [10], without clear understanding that the tuples must be right associating.

2.1.4 Ad hoc Association

There are two other ways to build a 4-tuple out of pairs, and using Sigma types we are free to use any ad hoc association we choose. However, it seems that unitary rules for building records, to be discussed below, must choose left or right association uniformly.

2.2 Inductive Telescopes

The inductive definition of Sigma types as 2-tuples, with projections programmed in terms of inductive elimination, can be extended to arbitrary telescopes. If

$$\mathbb{T} \;\equiv\; [x_1\!: A_1][x_2\!: A_2(x_1)] \cdots [x_k\!: A_k(x_1, \ldots, x_{k-1})]$$

is a telescope, then there is an inductively defined type $\Sigma\mathbb{T}$, with a single constructor, given by the formation and introduction rules

$$\textsc{Form}\; \frac{\begin{array}{l} A_1 : \mathtt{Type} \\ A_2 : (A_1)\mathtt{Type} \\ \vdots \\ A_k : (x_1\!: A_1;\ x_2\!: A_1(x_1);\ \ldots;\ A_{k-1}(x_1, \ldots, x_{k-2}))\mathtt{Type} \end{array}}{\Sigma\mathbb{T} : \mathtt{Type}} \qquad (9)$$

$$\textsc{Intro}\; \frac{\Sigma\mathbb{T} : \mathtt{Type} \quad a_1 : A_1 \quad a_2 : A_2(a_1) \quad \cdots \quad a_k : A_k(a_1, \ldots, a_{k-1})}{\langle a_1, \ldots, a_k \rangle_{\Sigma\mathbb{T}} : \Sigma\mathbb{T}} \qquad (10)$$

The premises of (9) say exactly that $\mathbb{T}$ is well-typed, and the premises of (10) say that $\langle a_1, \ldots, a_k \rangle$ fits in to $\mathbb{T}$. In *Coq* (and similarly in *Lego*) there is special syntax for inductive telescopes. For example, one can define *PER* as

```
Record PER: Type :=
       { S:Set; R:S->S->Prop; Sym:(sym S R); Trn:(trn S R) }.
```

which generates an inductive definition

```
Inductive PER : Type :=
      Build_PER : (S:Set; R:S->S->Prop)(trn S R)->(sym S R)->PER.
```

having one constructor, `Build_PER`. The field names, `S`, `R`, ..., are bound variables. To access the fields from outside the package, one must use the anonymous eliminator `Cases`. For example, `S` is defined (approximately) as

```
S [x:PER]: Set := Cases x of (Build_PER S R _ _) => S.
```

Coq tries to define these projectors automatically, and name them with the associated bound variable name, but this will fail if that name is already used at top-level.

2.2.1 Uniform Packages

For every k we can define the general telescope of length k. Σ (Sect. 2.1) is the general 2-telescope. The general 4-telescope is programmed in *Coq* as

```
Record Sig4 [A1: Type;
             A2: A1->Type;
             A3: (a1:A1)(A2 a1)->Type;
             A4: (a1:A1; a2:(A2 a1))(A3 a1 a2)->Type] : Type :=
  { a1:A1; a2:(A2 a1); a3:(A3 a1 a2); a4:(A4 a1 a2 a3) }.
```

This definition exactly captures rules, (9) and (10). *PER* can be defined as

```
Definition PER': Type :=
     (Sig4 Set
           [S:Set](S->S->Prop)
           [S:Set; R:S->S->Prop](sym S R)
           [S:Set; R:S->S->Prop; _:(sym S R)](trn S R)).
```

It is also possible to define general telescopes in terms of shorter general telescopes in various ways, as we defined *PER* in several ways in terms of pairs.

2.2.2 Some Differences.

`PER` above is not heavily typed, since `Build_PER` can only construct a `PER`. E.g. `(Build_PER T Q symQ trnQ)` inhabits `PER`. On the other hand, `PER'` is heavily typed, and its constructor must be given type annotations.

Name equality vs structure equality. Due to the way *Coq* and *Lego* create inductive types, `PER` is different from any other type, so can be made abstract by hiding its constructor, as is done in programming language module systems. However `PER'` is definitionally equal to any type that is `Sig4` applied to arguments that convert with those in the definition of `PER'`. To make `PER'` abstract would require hiding the `Sig4` constructor. Similarly, any structure defined by specializing a more general package admits extra type equalities.

3 Dependently Typed Records

Betarte and Tasistro [4, 3, 19] give an extension of Martin-Löf's framework to extensible, dependently typed records, with record subtyping. They had a prototype implementation, and worked interesting examples. However their presentation, in a complicated framework, with apparently essential use of subtyping, makes their system hard to understand. Here I give

$$\textsc{Form}\ \frac{L : \texttt{Type} \qquad A : (L)\texttt{Type}}{\langle L,\ \mathsf{r}\!: A\rangle : \texttt{Type}} \qquad\qquad \textsc{Intro}\ \frac{\langle L,\ \mathsf{r}\!: A\rangle : \texttt{Type} \qquad l : L \qquad a : A(l)}{\langle l,\ \mathsf{r}\!=a\rangle : \langle L,\ \mathsf{r}\!: A\rangle}$$

Elimination: projection and restriction of the visible label.

$$\textsc{El-tr}\ \frac{l : \langle L,\ \mathsf{r}\!: A\rangle}{l|\mathsf{r} : L} \qquad\qquad \textsc{El-tp}\ \frac{l : \langle L,\ \mathsf{r}\!: A\rangle}{l.\mathsf{r} : A(l|\mathsf{r})}$$

Elimination: passing the operations below top level.

$$\textsc{El-lr}\ \frac{l : \langle L,\ \mathsf{r}\!: A\rangle \qquad (l|\mathsf{r})|\mathsf{p} : P}{l|\mathsf{p} : P}\ \mathsf{r} \neq \mathsf{p} \qquad\qquad \textsc{El-lp}\ \frac{l : \langle L,\ \mathsf{r}\!: A\rangle \qquad (l|\mathsf{r}).\mathsf{p} : P}{l.\mathsf{p} : P}\ \mathsf{r} \neq \mathsf{p}$$

Computation rules.

$$\textsc{C-tr}\ \frac{\langle l,\ \mathsf{r}\!=a\rangle : \langle L,\ \mathsf{r}\!: A\rangle}{\langle l,\ \mathsf{r}\!=a\rangle|\mathsf{r} = l : L} \qquad\qquad \textsc{C-tp}\ \frac{\langle l,\ \mathsf{r}\!=a\rangle : \langle L,\ \mathsf{r}\!: A\rangle}{\langle l,\ \mathsf{r}\!=a\rangle.\mathsf{r} = a : A(l)}$$

$$\textsc{C-lr}\ \frac{l : \langle L,\ \mathsf{r}\!: A\rangle \qquad (l|\mathsf{r})|\mathsf{p} : P}{(l|\mathsf{r})|\mathsf{p} = l|\mathsf{p} : P}\ \mathsf{r} \neq \mathsf{p} \qquad\qquad \textsc{C-lp}\ \frac{l : \langle L,\ \mathsf{r}\!: A\rangle \qquad (l|\mathsf{r}).\mathsf{p} : P}{(l|\mathsf{r}).\mathsf{p} = l.\mathsf{p} : P}\ \mathsf{r} \neq \mathsf{p}$$

Figure 2: Rules for left associating records.

a simplified version of their system, without subtyping or explicit substitutions, which turns out to be straightforward left associating records. (I am informal about the difference between `Type` and `Set`.) Record subtyping will be treated orthogonally (using *coercive subtyping*) in Sect. 4. I also present right associating records, which are reasonable too.

3.1 Left Associating Records

A left associating record is a pair, with a labelled second component. It responds only to its own label, but passes any other label on to its first component. *Record types* (*signatures*) have syntax $\langle L,\ \mathsf{r}\!: A\rangle$. *Records* (*structures*) have syntax $\langle l,\ \mathsf{r}\!=a\rangle_{\langle L,\ \mathsf{r}: A\rangle}$. I will henceforth omit type annotations on records, which are, however, necessary in this setting for type synthesis to be effective. The rules are shown in Fig. 2.

Labels do not bind, either in signatures or in records. In the notation $\langle l,\ \mathsf{r}\!=a\rangle$, "$\mathsf{r}\!=a$" is only syntax, not a local definition. It is more precise to say that the label is part of the signature, hence of the type annotation of a record.

Formation and Introduction In rules Form and Intro, L may be a record type, and may have a field `r`. Later fields shadow earlier fields with the same label. Allowing repeated labels in signatures is not completely satisfactory, but simplifies the presentation, while allowing signatures to be first-class.[3] Betarte and Tasistro use a kind, *record-type*, to enforce that if $\rho :$ *record-type* is known, then all the labels of ρ are known, so side conditions about freshness of labels make sense. Our presentation could be modified to enforce fresh labels in a similar way.

[3]E.g. $[L\!: \texttt{Type},\ A\!: (L)\texttt{Type}]\langle L,\ \mathsf{r}\!: A\rangle$ is well-typed, although L may have label `r`.

$$\text{S-TR}\ \frac{\langle L,\ \mathsf{r}\colon A\rangle : \mathtt{Type}}{\langle L,\ \mathsf{r}\colon A\rangle|\mathsf{r} = L : \mathtt{Type}} \qquad \text{S-TP}\ \frac{\langle L,\ \mathsf{r}\colon A\rangle : \mathtt{Type}}{\langle L,\ \mathsf{r}\colon A\rangle.\mathsf{r} = A : (L)\mathtt{Type}}$$

$$\text{S-LR}\ \frac{\langle L,\ \mathsf{r}\colon A\rangle : \mathtt{Type} \quad L|\mathsf{p} : \mathtt{Type}}{\langle L,\ \mathsf{r}\colon A\rangle|\mathsf{p} = L|\mathsf{p} : \mathtt{Type}}\ \mathsf{r} \neq \mathsf{p} \quad \text{S-LP}\ \frac{\langle L,\ \mathsf{r}\colon A\rangle : \mathtt{Type} \quad L|\mathsf{p} : \mathtt{Type}}{\langle L,\ \mathsf{r}\colon A\rangle.\mathsf{p} = L.\mathsf{p} : (L|\mathsf{p})\mathtt{Type}}\ \mathsf{r} \neq \mathsf{p}$$

Figure 3: Signature operations for left associating records.

Similarly, for us no empty record type is required; an informal notion of "pure record type" is obtained by starting from the unit type (which I write as $\langle\rangle$, although it is not a record).

Elimination and Computation There are two record eliminators, projection "$l.\mathsf{r}$" and restriction "$l|\mathsf{r}$". The side condition $\mathsf{r} \neq \mathsf{p}$ is needed to force the top-level rules to be used whenever possible, as r may be a field in $l|\mathsf{r}$. In rules EL-LR (resp. EL-LP) we don't know that L is a record type, but $l|\mathsf{p}$ (resp. $l.\mathsf{p}$) is only well typed if we already know that $(l|\mathsf{r})|\mathsf{p}$ (resp. $(l|\mathsf{r}).\mathsf{p}$) is well typed, which can only happen if L is (equal to) a record type with a field labelled p.

Betarte and Tasistro use record subtyping to explain projection: if $l : L$, and $\mathsf{r}\colon A$ is a field in L, they conclude $l.\mathsf{r} : A(l)$. That is, the only use A can make of l is to project it at some fields that precede r in L, and l has at least those fields that precede r. Our restriction operation, which is not explicit for Betarte and Tasistro, allows us to write the elimination rules (e.g. EL-TP) prior to a notion of record subtyping.

The computation rules are as expected, given the elimination rules. I have written them as typed equality. While rules C-TR and C-TP can be implemented by syntactic reduction, rules C-LR and C-LP apparently require run-time type checking because of their second premises.

Records vs Sigma It is clear from our presentation that left associating signatures are just Sigma types carrying labels: erase "r" everywhere and read "|" and "." as ".1" and ".2" respectively. Thus they can be nested however you wish; we call them *left associating* because rule EL-LP searches from right to left. They could have an anonymous restriction operator (just drop rule EL-LR), but whenever we know that some object has record type, we already know its label.

Aside: Operations on signatures In Fig. 2 I have given conventional rules EL-TR, ..., EL-LP, for elimination of a (hypothetical) object, l, of record type. In fact the object, l, plays no role, and alternative rules for elimination and computation on signatures can be given (Fig. 3). These rules are slightly sharper than those in Fig. 2, and I use them in Sect. 6. Elimination of record objects is easily expressed in terms of these signature operations.

3.1.1 Example: *PER* in left associating records

In "official" syntax we write

$$\begin{aligned} \mathit{set} &:= \langle\langle\rangle,\ \mathsf{S}\colon[\_\colon\langle\rangle]\mathtt{Set}\rangle \\ \mathit{Rel} &:= \langle \mathit{set},\ \mathsf{R}\colon[x\colon \mathit{set}](x.\mathsf{S})(x.\mathsf{S})\mathtt{Prop}\rangle \\ \mathit{symRel} &:= \langle \mathit{Rel},\ \mathsf{sAx}\colon[x\colon \mathit{Rel}]\mathit{sym}(x.\mathsf{S})(x.\mathsf{R})\rangle \\ \mathit{PER} &:= \langle \mathit{symRel},\ \mathsf{tAx}\colon[x\colon \mathit{symRel}]\mathit{trn}(x.\mathsf{S})(x.\mathsf{R})\rangle \end{aligned}$$

Binding of variables for local field names is treated the same as in (7), although here, because of rules C-LR and C-LP, the depth of projections does not increase ("$\mathit{trn}(x.\mathsf{S})(x.\mathsf{R})$" instead of "$\mathit{trn}(P.1.1)(P.1.2)$"). The field labels are new, but they do not interact with local names. This structure could be defined with three nested pairs, as in (7), but then the first field would not have a label; it would still be accessible as $l|\mathsf{R}$.

Writing this signature without intermediate definitions, and removing inner brackets and redundant type tags, we get a nearly acceptable notation:

$$\begin{aligned} \mathit{PER} \quad := \quad & \langle \mathsf{S}\colon[\_]\mathtt{Set},\ \mathsf{R}\colon[x](x.\mathsf{S})(x.\mathsf{S})\mathtt{Prop},\ \mathsf{sAx}\colon[x]\mathit{sym}(x.\mathsf{S})(x.\mathsf{R}), \\ & \mathsf{tAx}\colon[x]\mathit{trn}(x.\mathsf{S})(x.\mathsf{R})\rangle. \end{aligned} \tag{11}$$

An official inhabitant of (11) is $\langle\langle\langle\langle\star,\ \mathsf{S}{=}T\rangle,\ \mathsf{R}{=}Q\rangle,\ \mathsf{sAx}{=}\,\mathit{sym}Q\rangle,\ \mathsf{tAx}{=}\,\mathit{trn}Q\rangle$, where $\star : \langle\rangle$. Omitting redundant information, we write

$$\langle \mathsf{S}{=}T,\ \mathsf{R}{=}Q,\ \mathsf{sAx}{=}\,\mathit{sym}Q,\ \mathsf{tAx}{=}\,\mathit{trn}Q\rangle. \tag{12}$$

3.2 Right Associating Records

I presented left associating records first, as they are close to the work of Betarte and Tasistro. However, from the viewpoint that records are just Sigma types with labels, right associating records are also natural. I guess that Betarte and Tasistro chose left association to achieve *extensibility*.

A right associating record is a pair, with a labelled first component. It responds only to its own label, but passes any other label on to its second component. *Record types* have syntax $\{\mathsf{r}\colon A,\ L\}$. *Records* have syntax $\{\mathsf{r}{=}a,\ l\}_{\{\mathsf{r}\colon A,\ L\}}$. I will henceforth omit type annotations on records, which are, however, necessary for type synthesis to be effective. Again, "$\mathsf{r}{=}\,a$" is not a local definition: r does not bind in l.

The rules are analogous to left associating records of Sect. 3.1, so I only present some of them in Fig. 4.

Signature abstraction Unlike the rules of Fig. 3, I do not see how to express elimination rules for right associating signatures, since here, carrying the operations underneath the top level involves a signature family, $L : (A)\mathtt{Type}$, rather than a signature. However, the top level signature operations are definable (rules S-TR and S-TP of Fig. 4). Restriction, $\{\mathsf{r}\colon A,\ L\}|$, takes a closed package, and shows how it functionally depends on its first field. This operation supports the *signature application* of [10], and hence Pebble-style sharing (Sects. 2.1.3 and 5.1). This is the main reason to be interested in right associating records.

$$\textsc{Form}\ \frac{A : \mathtt{Type} \qquad L : (A)\mathtt{Type}}{\{\mathsf{r}\colon A,\ L\} : \mathtt{Type}} \qquad\qquad \textsc{Intro}\ \frac{\{\mathsf{r}\colon A,\ L\} : \mathtt{Type} \qquad a : A \qquad l : L(a)}{\{\mathsf{r}{=}\, a,\ l\} : \{\mathsf{r}\colon A,\ L\}}$$

$$\textsc{El-tr}\ \frac{l : \{\mathsf{r}\colon A,\ L\}}{l|\mathsf{r} : L(l.\mathsf{r})} \qquad\qquad \textsc{El-tp}\ \frac{l : \{\mathsf{r}\colon A,\ L\}}{l.\mathsf{r} : A}$$

$$\textsc{El-lr}\ \frac{l : \{\mathsf{r}\colon A,\ L\} \qquad (l|\mathsf{r})|\mathsf{p} : P}{l|\mathsf{p} : P}\ \mathsf{r} \neq \mathsf{p} \qquad\qquad \textsc{El-lp}\ \frac{l : \{\mathsf{r}\colon A,\ L\} \qquad (l|\mathsf{r}).\mathsf{p} : P}{l.\mathsf{p} : P}\ \mathsf{r} \neq \mathsf{p}$$

Signature abstraction:

$$\textsc{S-tr}\ \frac{\{\mathsf{r}\colon A,\ L\} : \mathtt{Type}}{\{\mathsf{r}\colon A,\ L\}| = L : (A)\mathtt{Type}} \qquad\qquad \textsc{S-tp}\ \frac{\{\mathsf{r}\colon A,\ L\} : \mathtt{Type}}{\{\mathsf{r}\colon A,\ L\}. = A : \mathtt{Type}}$$

Figure 4: Some rules for right associating records.

3.2.1 Example: *PER* in right associating records

In "official" syntax we write

$$\begin{aligned} \mathit{PER} \ := \quad & \{\mathsf{S}\colon \mathtt{Set},\ [s\colon \mathtt{Set}] \\ & \quad \{\mathsf{R}\colon (s)(s)\mathtt{Prop},\ [r\colon (s)(s)\mathtt{Prop}] \\ & \qquad \{\mathsf{sAx}\colon \mathit{sym}(s)(r),\ [\_\colon \mathit{sym}(s)(r)] \\ & \qquad\quad \{\mathsf{tAx}\colon \mathit{trn}(s)(r),\ [\_\colon \mathit{trn}(s)(r)]\langle\rangle\}\}\}\}. \end{aligned}$$

Removing inferable type annotations and internal brackets, we can write

$$\mathit{PER} := \{\mathsf{S}\colon \mathtt{Set},\ [s]\mathsf{R}\colon (s)(s)\mathtt{Prop},\ [r]\mathsf{sAx}\colon \mathit{sym}(s)(r),\ [\_]\mathsf{tAx}\colon \mathit{trn}(s)(r)\}. \tag{13}$$

Binding of variables for local names is treated the same as in (4). An official inhabitant of (13) is $\{\mathsf{S}{=}\,T,\ \{\mathsf{R}{=}\,Q,\ \{\mathsf{sAx}{=}\,\mathit{sym}Q,\ \{\mathsf{tAx}{=}\,\mathit{trn}Q,\ \star\}\}\}\}$. Omitting redundant information, we write $\{\mathsf{S}{=}\,T,\ \mathsf{R}{=}\,Q,\ \mathsf{sAx}{=}\,\mathit{sym}Q,\ \mathsf{tAx}{=}\,\mathit{trn}Q\}$.

3.3 Labels and Variables

Labels are the global accessors for records, and hence cannot be alpha converted. Thus labels and variables cannot be the same syntactic class in dependent record types, or else how could we substitute y for x in $\langle y\colon \mathtt{Set},\ z\colon x\rangle$. To my knowledge, the first satisfactory handling of this problem is in [9], where every field has both a label (that does not bind) and a variable (that does bind). *PER* would be written as

$$\langle \mathsf{S} \triangleright s\colon \mathtt{Set},\ \mathsf{R} \triangleright r\colon (s)(s)\mathtt{Prop},\ \mathsf{sAx} \triangleright \_\colon \mathit{sym}(s)(r),\ \mathsf{tAx} \triangleright \_\colon \mathit{trn}(s)(r)\rangle. \tag{14}$$

The approaches I give above, inspired by [4], are more parsimonious than [9] by using the existing dependent function type instead of introducing a new binding construct. Nonetheless, all three notations with labels and variables (i.e. (11), (13) and (14)) are heavy. Betarte and Tasistro point out that an informal notation, e.g.

$$\langle \mathsf{S}\colon \mathtt{Set},\ \mathsf{R}\colon (\mathsf{S})(\mathsf{S})\mathtt{Prop},\ \mathsf{sAx}\colon \mathit{sym}(\mathsf{S})(\mathsf{R}),\ \mathsf{tAx}\colon \mathit{trn}(\mathsf{S})(\mathsf{R})\rangle, \tag{15}$$

can be translated mechanically to the formal notations. This is true, but some signatures cannot be represented in this informal manner, as an example from [9] shows:

$$\begin{array}{ll}
\text{left associating} & \langle \mathsf{b}\colon [_\colon \langle\rangle]\mathtt{Type},\ \mathsf{c}\colon [x]\langle \mathsf{b}\colon [_\colon \langle\rangle]\mathtt{Type},\ \mathsf{f}\colon [y](y.\mathsf{b})x.\mathsf{b}\rangle\rangle, \\
\text{right associating} & \{\mathsf{b}\colon \mathtt{Type},\ [x\colon \mathtt{Type}]\{\mathsf{b}\colon \mathtt{Type},\ [y\colon \mathtt{Type}]\mathsf{f}\colon (y)x\}\}, \\
\text{Harper/Lillibridge} & \langle \mathsf{b} \triangleright x\colon \mathtt{Type},\ \mathsf{c} \triangleright _\colon \langle \mathsf{b} \triangleright y\colon \mathtt{Type},\ \mathsf{f} \triangleright _\colon (y)x\rangle\rangle.
\end{array}$$

Here, the variables x, y, are needed to disambiguate the two labels b. Thus it seems unlikely that a notation such as (15) could be a satisfactory formalization. Nonetheless, some well known proof tools (e.g. *PVS*) present dependent record types in this notation.

In the three formal approaches, left- and right- associating records, and Harper/Lillibridge style, labels and variables are distinct syntactic classes. But there can be no confusion, even if they are implemented with the same concrete type, as labels and variables only appear in different syntactic positions.

3.4 Dependent Records?

Both Harper/Lillibridge and Courant [6] present structures that are *dependent* as well as *dependently typed*; i.e., field variables can be used in later fields as local definitions. For example, one could write $\langle \mathsf{n} \triangleright x{=}3,\ \mathsf{m} \triangleright \_{=}x\rangle$.[4] However, this record is definitionally equal to $\langle \mathsf{n} \triangleright x{=}3,\ \mathsf{m} \triangleright \_{=}3\rangle$, so no judgement can distinguish between the dependent presentation and the flattened one.

It is evident from rules Intro of Figs. 2 and 4 that fields in my left and right associating records cannot depend on previous fields. But my records are equally expressive: I write $\langle \mathsf{n}{=}3,\ \mathsf{m}{=}3\rangle$ at the same type as the example above. To preserve sharing, I might take $\langle \mathsf{n}{=}3,\ \mathsf{m}{=}\mathsf{n}\rangle$ as syntactic sugar for *let* $n = 3$ *in* $\langle \mathsf{n}{=}n,\ \mathsf{m}{=}n\rangle$, where n is a local variable that must be sufficiently fresh. Nonetheless, the dependency in Harper/Lillibridge structures does seem useful, even if I cannot say precisely what the difference is. I guess it is relevant here to mention that Harper/Lillibridge structures are not extensible: once a structure is closed there is no "potential" future use of its field variables.

4 Coercions

Presentations such as [9, 4, 6] have a built-in notion of *structural subtyping*: roughly, any well-typed permutation of the fields of an extension of a signature is a subtype of that signature. Further, one allows corresponding fields themselves to be in the subtype relation. Whenever an object of a certain type is required, an object of a subtype is accepted: e.g. you can use a group wherever a monoid is expected. Below, I show how this approach is limited when structures are not "flat", but are constructed from substructures, as is desirable in practice.

Peter Aczel [1] suggested a notion of *coercive subtyping* for type theory. Both *Coq* and *Lego* now support ad hoc, but very useful, notions of coercive subtyping [18, 2]. Zhaohui Luo and colleagues [14, 15] have studied coercive subtyping foundationally, as an abbreviational mechanism which introduces definitional equalities at a logical framework level. This approach is more expressive in some ways than the implementations, but in other respects does not yet equal them. Also, we do not claim that all the usability and implementability issues of the logical framework (typed equality judgement, universes á la Tarski, etc.) are worked out.

[4] This style requires field variables in structures as well as in signatures.

As an example of coercive subtyping, reconsider equation (5), using coercions in *Coq* notation. Define `Inner` as a parameterized record, and use it as a field in a structure for `PER`

```
Record Inner [S:Set; R:S->S->Prop]: Type :=
  { Sym: (sym S R); Trn: (trn S R) }.
Record PER: Type := { S:>Set; R:>S->S->Prop; i:>(Inner S R) }.
```

The notation `S:>Set` means that `S:PER->Set` is treated as a coercion from `PER` to `Set`, so that if `p:PER` we can write `x:p` to mean `x` ranging over the carrier of `p`, i.e. `x:(S p)`, with the typechecker invisibly inserting the coercion `S`. Similarly `R:(p:PER)p->p->Prop` is a coercion from `p:PER` to relations over the carrier of `p`. (We can write `p` instead of `(S p)` in the type of `R`.) The statement about a particular `p:PER` that its relation is reflexive can be written as `(x:p)(p x x)`.

Similarly `i:>(Inner S R)` means that `i:(p:PER)(Inner (S p) (R p))` (which can be written as `i:(p:PER)(Inner p)`) is treated as a coercion from `PER` to `Inner`, so that whenever `p:PER`, the field projections `Sym` and `Trn` of `Inner` can be applied directly to `p`, with the typechecker invisibly inserting the coercion `i`. Thus `PER` appears to be a subtype of `Inner`.[5]

Structural subtyping cannot show `PER` as a subtype of `Inner`, because `PER` is not a permutation of an extension of `Inner`. In practice we often build new structures out of existing structures in this way (e.g. *ringSig* in Sect. 5.1). Structural subtyping depends on accidents of structure, and does not support natural mathematical definitions.

In this example the coercion `i` *open*s the `Inner` structure for users of the structure `PER`. This opening is transitive, in the sense that if we use `PER` in a larger structure, e.g.

```
Record ER: Type := { p:>PER; Rfl:(rfl p) }.
```

then we can still project the fields of `Inner` from an `ER` object. This also shows that extensibility is not very important, given coercive subtyping.

In practice we want *renaming* to support opening substructures with common labels; e.g. the additive and multiplicative monoids of a ring.

4.1 Subtyping Sigma Types

Since records are constructed like nested Sigma types, it is useful to ask how subtyping propagates through Sigma. Suppose L is a subtype of L' by coercion $c : (L)L'$, and $A : (L')\mathtt{Type}$. c can be lifted to a coercion $c_\Sigma : (\Sigma\, L\, A{\circ}c)\Sigma\, L'\, A$ showing that $\Sigma\, L\, A{\circ}c$ is a subtype of $\Sigma\, L'\, A$, defined by $c_\Sigma(\langle l, a\rangle) := \langle c(l), a\rangle$ [14]. Note how the composition, $A{\circ}c$, is needed for the subtype to be well formed, as A expects an L'-object, not an L-object. This idiom will appear again in Sect. 5.4.

5 Manifest Signatures

Programming language designers have long recognized the need to see through the signatures of modules; [11, 16, 13, 9] give a taste of the relevant literature. I assume the reader is familiar with the two basic approaches, *Pebble style sharing* which uses pure abstraction and application, and *manifest types* in signatures, which requires some new explanation. (I use the term *manifest types*, from [13], informally, not referring to particulars of that paper.)

[5]The illusion is not complete, as `Inner` is parameterized, and the projections must be applied to the parameters: `(Trn (S p) (R p) p)`. However, with *implicit arguments*, we can write `(Trn p)`.

5.1 Why Manifest Signatures?

Expressive Theorems With any of the packages above, the lemma that the dual of a PER is also a PER could be stated as $(PER)PER$, but this formulation is too coarse to express our meaning. E.g. the identity function is also a proof of $(PER)PER$ but is not the duality construction.

We have seen the application of a parameterized signature (*Pebble-style* sharing) in Sect. 2.1.3. This is convenient for right associating structures [10]. Using (5) as the definition of PER, the duality theorem can be stated as

$$(p\colon PER)Inner(p.\mathsf{S})(dual(p.\mathsf{S})(p.\mathsf{R})).$$

This shows the duality explicitly, but doesn't actually return a PER, so operations on the PER package cannot be applied to the structure returned. E.g. we cannot use this theorem directly to prove that dualization is involutive.

A manifest signature expresses the intended meaning

$$(p\colon PER)\langle \mathsf{S}{=}\,p.\mathsf{S}\colon \mathtt{Set},\ \mathsf{R}{=}\,dual(\mathsf{S})(p.\mathsf{R})\colon (\mathsf{S})(\mathsf{S})\mathtt{Prop},\ \mathsf{sAx}\colon sym(\mathsf{S})(\mathsf{R}),\ \ldots\rangle \tag{16}$$

but forces us to rewrite the definition of PER, which is error prone and obscures the statement. (Since S is manifest in (16), we use S in place of its value in succeeding fields.)

What is needed is something like the *with* notation [13] to add information about a signature. The duality lemma could be stated as

$$(p\colon PER)PER\ \mathit{with}\ \mathsf{S}{=}\,p.\mathsf{S}\ \mathit{with}\ \mathsf{R}{=}\,dual(\mathsf{S})(p.\mathsf{R}). \tag{17}$$

This is often considered syntactic sugar for (16), but there are some details to make precise (Sect. 5.4.1).

This example shows that ordinary theorems involve functions from structures to structures. Thus I am interested in first-class records with manifest types, and guess that stratified module systems [6] will be inadequate in practice.

Sharing Suppose *monSig* is the signature of monoids, and *grpSig* is the signature of groups. How can we define *ringSig* as an (additive) group and a (multiplicative) monoid, sharing the same carrier set, and having some axioms connecting the two operations? We might be satisfied with *Pebble-style* sharing, "applying" *monSig* to the carrier of the group. Although the objection still arises that in this approach no multiplicative monoid actually occurs as a substructure of a ring, attempts to formalize some algebra in this way have made interesting progress [17]. Nonetheless, *Pebble-style* sharing is heavy, and doesn't scale up in practice. What is needed is the *with* notation, allowing to write

$$ringSig \quad := \quad \langle \mathsf{G}\colon grpSig,\ \mathsf{M}\colon monSig\ \mathit{with}\ \mathsf{crr}{=}\,\mathsf{G.crr},\ \ldots\rangle.$$

5.2 Definable Manifest Signatures

5.2.1 Left Associating

Using single constructor inductive definitions we can add a value specification to the right field of a Sigma type (Fig. 5). We call these *left associating* only because we intend to use them that way, as the example below shows. ΨLAa doesn't say what the value of the second

$$\textsc{Form}\ \frac{\Sigma LA : \mathtt{Type} \qquad a : (l{:}\,L)A(l)}{\Psi LAa : \mathtt{Type}} \qquad\qquad \textsc{Intro}\ \frac{\Psi LAa : Type \qquad l : L}{\langle l\rangle_{\Psi LAa} : \Psi LAa}$$

Computation rules.

$$\textsc{Comp1}\ \frac{\langle l\rangle_{\Psi LAa} : \Psi LAa}{\langle l\rangle_{\Psi LAa}.1 = l : L} \qquad\qquad \textsc{Comp2}\ \frac{l : \Psi LAa}{l.2 = a(l.1) : A(l.1)}$$

Figure 5: Some rules for Ψ, manifest left associating pairs.

field is, but constrains it uniformly as a function of the first field. As before, I will write ΨAa instead of ΨLAa, and even more informally, $\langle l\rangle_a$ for $\langle l\rangle_{\Psi Aa}$.

In *Coq* notation this type is defined

```
Record Psi [L:Type; A:L->Type; a:(l:L)(A l)]: Type := { psi1:L }.
```

The first projection, rule Comp1, is defined by inductive elimination as before, but the second projection, rule Comp2, is defined using the heavy type annotations

```
Definition psi2 [L:Type; A:L->Type; a:(l:L)(A l); h:(Psi ?? a)]
       : (A (psi1 ??? h)) := (a (psi1 ??? h)).
```

This strong rule shows that ΨAa has its second field manifest.

From Intro it is clear that ΨAa is isomorphic with L. From Form, it is clear that ΨAa is a "subtype" of ΣA; there is a definable coercion that makes this subtyping implicit in *Coq*

```
Definition Psi_sigT [L:Type; A:L->Type; a:(l:L)(A l); h:(Psi ? A a)]
  : (sigT ? A) := (existT ?? (psi1 ??? h) (psi2 ??? h)).
Coercion Psi_sigT: Psi >-> sigT.
```

Example Surprisingly, although every ΨAa is uniformly typed (i.e. $\Psi Aa : \mathtt{Type}$ implies $A : (L)\mathtt{Type}$), it is possible to construct non-uniform signatures. For example, although the informal signature $\langle K{:}\,\mathtt{Type},\ A{:}\,K,\ a{:}\,A\rangle$ is not well-typed (e.g. if $K{=}\,(\mathtt{Set})\mathtt{Set}{:}\,\mathtt{Type}$), the manifest signature $\langle K{=}\,\mathtt{Set}{:}\,\mathtt{Type},\ A{:}\,K,\ a{:}\,A\rangle$ is well-typed by the construction

$$\begin{array}{lcll} \mathit{L1} & := & \Psi([\_{:}\,\langle\rangle]\mathtt{Type})([\_{:}\,\langle\rangle]\mathtt{Set}) & \text{representing } \langle\langle\rangle,\ K{=}\,\mathtt{Set}{:}\,\mathtt{Type}\rangle, \\ \mathit{L2} & := & \Sigma[x{:}\,\mathit{L1}]x.2 & \text{representing } \langle L1,\ A{:}\,K\rangle, \\ \mathit{L3} & := & \Sigma[x{:}\,\mathit{L2}]x.2 & \text{representing } \langle L2,\ a{:}\,A\rangle. \end{array} \tag{18}$$

By rule Comp2, *L2* is definitionally equal to $\Sigma[x{:}\,\mathit{L1}]\mathtt{Set}$ (informally $\langle L1,\ A{:}\,\mathtt{Set}\rangle$). *L3* is inhabited by the tuple $\langle\langle\langle\star\rangle_{[\_{:}\,\langle\rangle]\mathtt{Set}},\ nat\rangle,\ 0\rangle$.

Example: *PER* duality It is not only pathalogical examples, such as the previous one, that require the manifest computation rule. The typing of (16) also depends on manifest computation, although the underlying, unmanifest, signature, *PER*, is well typed. Consider three variations of (16):

$$(p{:}\,\mathit{PER})\langle \mathsf{S}{:}\,\mathtt{Set},\ \mathsf{R}{=}\,\mathit{dual}(\mathsf{S})(p.\mathsf{R}){:}\,(\mathsf{S})(\mathsf{S})\mathtt{Prop},\ \mathsf{sAx}{:}\,\mathit{sym}(\mathsf{S})(\mathsf{R}),\ \ldots\rangle, \tag{19}$$

$$(p{:}\,\mathit{PER})\langle \mathsf{S}{:}\,\mathtt{Set},\ \mathsf{R}{=}\,\mathit{dual}(p.\mathsf{S})(p.\mathsf{R}){:}\,(p.\mathsf{S})(p.\mathsf{S})\mathtt{Prop},\ \mathsf{sAx}{:}\,\mathit{sym}(p.\mathsf{S})(\mathsf{R}),\ \ldots\rangle, \tag{20}$$

$$(p{:}\,\mathit{PER})\langle \mathsf{S}{=}\,p.\mathsf{S}{:}\,\mathtt{Set},\ \mathsf{R}{=}\,\mathit{dual}(p.\mathsf{S})(p.\mathsf{R}){:}\,(p.\mathsf{S})(p.\mathsf{S})\mathtt{Prop},\ \mathsf{sAx}{:}\,\mathit{sym}(p.\mathsf{S})(\mathsf{R}),\ \ldots\rangle. \tag{21}$$

$$\textsc{Form}\ \frac{\Sigma AL : \texttt{Type} \qquad a : A}{\Phi AaL : \texttt{Type}} \qquad\qquad \textsc{Intro}\ \frac{\Phi AaL : Type \qquad l : L(a)}{\langle l\rangle_{\Phi AaL} : \Phi AaL}$$

$$\textsc{Comp1}\ \frac{l : \Phi AaL}{l.1 = a : A} \qquad\qquad \textsc{Comp2}\ \frac{\langle l\rangle_{\Phi AaL} : \Phi AaL}{\langle l\rangle_{\Phi AaL}.2 = l : L(a)}$$

Figure 6: Some rules for Φ, manifest right associating pairs.

Equation (19) is not well typed, since the dependency of *dual* requires that $dual(\mathsf{S})(p.\mathsf{R})$ convert with $dual(p.\mathsf{S})(p.\mathsf{R})$. As in the pathalogical example above, this is accomplished by adding manifest information, in this case arriving at (16).

On the other hand, (20) is well typed, but does not express duality of PER. The range of (20) is not a subtype of PER, as its relation is not constrained to act on its set S. E.g. $\langle \text{bool}, =_{\text{nat}}, \ \ldots\rangle$ inhabits the range of (20), but is not a PER.

Equation (21) is convertible with (16), and no type theory judgement can distinguish them.

5.2.2 Other Definable Manifest Telescopes

The same trick can be used to make fields manifest in any inductive telescope. Figure 6 shows how to make the left field of a Σ type manifest. (I will write ΦaL instead of ΦAaL.) Analogous to Ψ, Φ is inductively definable in type theory, and the representation has the strong manifest computation rule Comp1. However, an inner signature must be closed before its manifest fields can be used in an outer signature. Thus, using Φ to build right associating structures, we can only represent the pathalogical example above by unfolding the manifest definition: $\Phi\ \texttt{Set}\ [K{:}\,\texttt{Type}](\Sigma\ [A{:}\,\texttt{Set}](\Sigma\ [a{:}\,A]\langle\rangle))$ is well typed, but $\Phi\ \texttt{Set}\ [K{:}\,\texttt{Type}](\Sigma\ [A{:}\,K](\Sigma\ [a{:}\,A]\langle\rangle))$ is not well typed, as we have only $K{:}\,\texttt{Type}$, rather than $x{:}\,L1$ as in (18). Similarly for the PER duality example. In this sense Ψ is better than Φ.

5.3 Subtyping Manifest Signatures

In the motivating example of PER duality, in order to view theorem (16) as returning a PER, I need a coercion that erases some manifest type information, so that the type of the structure returned in (16) can be seen as a subtype of PER. We have just seen the coercion `Psi_sigT` that does this for Ψ, returning the corresponding Σ. However there can be no coercion that allows viewing $\langle K{=}\texttt{Set}{:}\,\texttt{Type},\ A{:}\,K,\ a{:}\,A\rangle$ as a subtype of $\langle K{:}\,\texttt{Type},\ A{:}\,K,\ a{:}\,A\rangle$, since the latter is not even well typed. The usual rule for subtyping manifest signatures (e.g. [9, 6]) has a premise requiring the (less manifest) supertype to be well typed.

In my presentation, the coercion forgetting a particular manifest field is constructed by applying `Psi_sigT` at that field (which actually forgets the manifest value, and is always well typed), then successively lifting the coercion through the following fields using c_Σ (Sect. 4.1). The typing of c_Σ checks at each stage that the supertype is well typed, and this approach can be seen as analysing the usual rule into two unitary rules `Psi_sigT` and c_Σ.

$$\text{MFORM}\ \frac{\langle L,\ \mathsf{r}{:}\,A\rangle : \texttt{Type} \quad a : (l{:}L)A(l)}{\langle L,\ \mathsf{r}{=}\,a{:}\,A\rangle : \texttt{Type}} \qquad \text{MINTRO}\ \frac{\langle L,\ \mathsf{r}{=}\,a{:}\,A\rangle : \texttt{Type} \quad l : L}{\langle l,\ \mathsf{r}{=}\,a(l)\rangle : \langle L,\ \mathsf{r}{=}\,a{:}\,A\rangle}$$

Manifest computation. $$\text{C-MAN}\ \frac{l : \langle L,\ \mathsf{r}{=}\,a{:}\,A\rangle}{l.\mathsf{r} = a(l|\mathsf{r}) : A(l|\mathsf{r})}$$

Figure 7: Some rules for manifest left associating records.

5.4 Manifest Left Associating Record Types

Just as we added labels to Σ in Sect. 3.1, so we add labels to Ψ. *Manifest* signatures have syntax $\langle L, \mathsf{r}{=}\,a{:}\,A\rangle$. r is *opaque* in $\langle L, \mathsf{r}{:}\,A\rangle$, and *manifest* in $\langle L, \mathsf{r}{=}\,a{:}\,A\rangle$. Objects inhabiting manifest signatures officially have syntax $\langle l\rangle_{\langle L,\, \mathsf{r}=\, a:\, A\rangle}$, but I will write them $\langle l, \mathsf{r}{=}\,a(l)\rangle$. Some rules are given in Fig. 7.

By inversion of MFORM, whenever $\langle L,\ \mathsf{r}{=}\,a{:}\,A\rangle$ is well typed, so is $\langle L,\ \mathsf{r}{:}\,A\rangle$. In general, however, well typedness of $\langle\langle L, r{=}a{:}A\rangle, s{:}B\rangle$ does not guarantee well typedness of $\langle\langle L, r{:}A\rangle, s{:}B\rangle$, as the typing of B in the former may depend on the value of r.

$\langle L,\ \mathsf{r}{=}\,a{:}\,A\rangle$ is more informative than $\langle L,\ \mathsf{r}{:}\,A\rangle$, so the elimination and computation rules of Sect. 3.1 also hold for $\langle L,\ \mathsf{r}{=}\,a{:}\,A\rangle$. We can define the coercion $(\langle L,\ \mathsf{r}{=}\,a{:}\,A\rangle)\langle L,\ \mathsf{r}{:}\,A\rangle$ (`Psi_sigT` in Sect. 5.2.1) from the projection and restriction operations of $\langle L,\ \mathsf{r}{=}\,a{:}\,A\rangle$. Thus $l : \langle L,\ \mathsf{r}{=}\,a{:}\,A\rangle$ can be used as if it had type $\langle L,\ \mathsf{r}{:}\,A\rangle$.

There is a new computation rule for the projection of a manifest field, C-MAN. It is admissible that rule C-MAN goes underneath the top constructor. For example, suppose $l : \langle L, \mathsf{p}{=}\,b{:}\,B, \mathsf{r}{:}\,A\rangle$; we have

$$\frac{\dfrac{\dfrac{l : \langle L, \mathsf{p}{=}\,b{:}\,B, \mathsf{r}{:}\,A\rangle}{l|\mathsf{r} : \langle L,\ \mathsf{p}{=}\,b{:}\,B\rangle}\ \text{EL-TR}}{(l|\mathsf{r}).\mathsf{p} = b((l|\mathsf{r})|\mathsf{p}) : B((l|\mathsf{r})|\mathsf{p})}\ \text{C-MAN} \qquad \begin{array}{c} l : \langle L, \mathsf{p}{=}\,b{:}\,B, \mathsf{r}{:}\,A\rangle \\ \vdots\ \text{C-LP} \\ (l|\mathsf{r}).\mathsf{p} = l.\mathsf{p} \end{array} \qquad \begin{array}{c} l : \langle L, \mathsf{p}{=}\,b{:}\,B, \mathsf{r}{:}\,A\rangle \\ \vdots\ \text{C-LR} \\ (l|\mathsf{r})|\mathsf{p} = l|\mathsf{p} \end{array}}{l.\mathsf{p} = b(l|\mathsf{p}) : B(l|\mathsf{p})}$$

5.4.1 The *with* Notation

"*with*" is explained in terms of manifest fields (Fig. 8). $\langle L,\ \mathsf{r}{:}\,A\rangle$ *with* $\mathsf{r}{=}\,a$ is defined as $\langle L,\ \mathsf{r}{=}\,a{:}\,A\rangle$ whenever the latter is well-typed (rule WITH-DEF). Having explained how to apply *with* to some record, L, we explain how to apply it to a longer record (rules WITH-RO and WITH-RM). Moving *with* to the right in a signature loses information: the typing of B in $\langle L$ *with* $\mathsf{r}{=}\,a,\ \mathsf{p}{:}\,B\rangle$ may use that $\mathsf{r}{=}\,a$, while in $\langle L,\ \mathsf{p}{:}\,B\rangle$ *with* $\mathsf{r}{=}\,a$ it may not. In Fig. 8 I do not allow *with* to over-write an already manifest field, but that alternative could be chosen as well.

Subtyping and *with* Since L *with* $\mathsf{r}{=}\,a$ is constructed by *adding* manifest information to a well typed signature, L, the difficulty of forgetting manifest information (Sect. 5.3) does not arise. There is a uniform coercion $u^L_{\mathsf{r}} : (L \textit{ with } \mathsf{r}{=}\,a)L$, which must be defined mutually with *with*, as it appears in the conclusions of rules WITH-RO and WITH-RM. The reason for this coercion in these rules is the lifting of subtyping through a signature, introduced in Sect. 4.1.

$$\textsc{With-def}\ \frac{\langle L,\ \mathsf{r}{=}\,a{:}\,A\rangle : \mathtt{Type}}{\langle L,\ \mathsf{r}{:}\,A\rangle\ \mathit{with}\ \mathsf{r}{=}\,a = \langle L,\ \mathsf{r}{=}\,a{:}\,A\rangle : \mathtt{Type}}$$

$$\textsc{With-ro}\ \frac{L\ \mathit{with}\ \mathsf{r}{=}\,a : \mathtt{Type} \qquad \langle L,\ \mathsf{p}{:}\,B\rangle : \mathtt{Type}}{\langle L,\ \mathsf{p}{:}\,B\rangle\ \mathit{with}\ \mathsf{r}{=}\,a = \langle L\ \mathit{with}\ \mathsf{r}{=}\,a,\ \mathsf{p}{:}\,B{\circ}u^L_{\mathsf{r}}\rangle : \mathtt{Type}}\ \mathsf{p} \neq \mathsf{r}$$

$$\textsc{With-rm}\ \frac{L\ \mathit{with}\ \mathsf{r}{=}\,a : \mathtt{Type} \qquad \langle L,\ \mathsf{p}{=}\,b{:}\,B\rangle : \mathtt{Type}}{\langle L,\ \mathsf{p}{=}\,b{:}\,B\rangle\ \mathit{with}\ \mathsf{r}{=}\,a = \langle L\ \mathit{with}\ \mathsf{r}{=}\,a,\ \mathsf{p}{=}\,b{\circ}u^L_{\mathsf{r}}{:}\,B{\circ}u^L_{\mathsf{r}}\rangle : \mathtt{Type}}\ \mathsf{p} \neq \mathsf{r}$$

Figure 8: Rules for *with*. $u^L_{\mathsf{r}} : (L\ \mathit{with}\ \mathsf{r}{=}\,a)L$ is the coercion forgetting the manifest information $\mathsf{r}{=}\,a$.

The uniform definition of u^L_{r} shows that *with*-signatures are well behaved with respect to subtyping. E.g. (17) obviously returns a subtype of PER, while this fact about (16) requires checking. (The check is the computation showing (17) and (16) to be definitionally equal.) Conversely, $\langle K{=}\,\mathtt{Set}{:}\,\mathtt{Type},\ A{:}\,K,\ a{:}\,A\rangle$ is not definitionally equal to any opaque signature followed by *with*s.

6 Programming Records in Type Theory

I have shown that dependently and manifestly typed structures can be "represented" in type theory using Σ and Ψ types (Sects. 2.1 and 5.2). That representation leaves labels in meta theory, outside the formalization. Also, signatures are not first class objects, so *with* can not be formalized either. To my surprise I found that dependently and manifestly typed structures, including labels and *with*, as described in Sect. 5.4, can be programmed in a logical framework such as that of Martin-Löf or Luo [14] extended with inductive-recursive definition [8]. The main reasons for doing so are to support experimentation with more interesting examples, to clarify details of the rules I have given above, and to gain confidence in the meta theory of my proposed system, which is now seen to be (in a sense) a definitional extension of LF plus inductive-recursive definition.

Neither *Lego* nor *Coq* officially supports the schema of induction-recursion, but I have implemented what follows in *Lego* by circumventing the schema checking.

Formation Let *lbl* be a type having decidable equality, e.g. *string*, to be used for labels. The type of signatures, $\mathit{sign} : \mathtt{Type}$, and for each signature, the type of records having that signature, $\mathit{recd} : (\mathit{sign})\mathtt{Type}$, are defined by induction-recursion. *sign* has introduction rules:

$$\textsc{Unit}\ \frac{}{\mathit{unSig} : \mathit{sign}} \qquad \textsc{Opaq}\ \frac{s : \mathit{sign} \quad \mathsf{r} : \mathit{lbl} \quad A : (\mathit{recd}(s))\mathtt{Type}}{\mathit{opSig}(s, \mathsf{r}, A) : \mathit{sign}}$$

$$\textsc{Man}\ \frac{s : \mathit{sign} \quad \mathsf{r} : \mathit{lbl} \quad A : (\mathit{recd}(s))\mathtt{Type} \quad a : (l{:}\mathit{recd}(s))A(l)}{\mathit{manSig}(s, \mathsf{r}, a, A) : \mathit{sign}}$$

For $s : sign$, $recd(s)$ computes the type of s-records (nested Σ and Ψ) by $sign$-elimination:[6]

$$\begin{array}{rcl} recd(unSig) &:& \texttt{Type} := \langle\rangle, \\ recd(opSig(s,\_,A)) &:& \texttt{Type} := \Sigma(recd(s),A), \\ recd(manSig(s,\_,a,A)) &:& \texttt{Type} := \Psi(recd(s),A,a). \end{array}$$

Here, as in the rules of previous sections, I have allowed labels to appear more than once in a signature. However, using induction-recursion, I could formalize a type of signatures without duplicate labels; [8] shows how this is done. In either case, I compute whether a label occurs in a signature by $sign$-elimination:

$$\begin{array}{rcl} occ(\mathsf{r}, unSig) &:& \texttt{Prop} := \texttt{False}, \\ occ(\mathsf{p}, opSig(s,\mathsf{r},\_)) &:& \texttt{Prop} := \textit{if } \mathsf{p}{=}\mathsf{r} \textit{ then } \texttt{True} \textit{ else } occ(\mathsf{p},s), \\ occ(\mathsf{p}, manSig(s,\mathsf{r},\_,\_)) &:& \texttt{Prop} := \textit{if } \mathsf{p}{=}\mathsf{r} \textit{ then } \texttt{True} \textit{ else } occ(\mathsf{p},s). \end{array}$$

Introduction and elimination Introduction operations are inherited from the underlying Σ and Ψ types, so need no further specification.

Given a proof of $occ(\mathsf{p},s)$, the operations of restriction and projection of signature s at label p can be defined by $sign$-elimination. Here, for convenience, I follow the alternative rules of Fig. 3, rather than those of Fig. 2.

$$\begin{array}{rcll} opSig(s,\mathsf{r},A)|\mathsf{p} &:& \texttt{Type} & := \textit{if } \mathsf{p}{=}\mathsf{r} \textit{ then } recd(s) \textit{ else } s|\mathsf{p} \\ opSig(s,\mathsf{r},A).\mathsf{p} &:& (opSig(s,\mathsf{r},A)|\mathsf{p})\texttt{Type} & := \textit{if } \mathsf{p}{=}\mathsf{r} \textit{ then } A \textit{ else } s.\mathsf{p} \\ manSig(s,\mathsf{r},A,a)|\mathsf{p} &:& \texttt{Type} & := \textit{if } \mathsf{p}{=}\mathsf{r} \textit{ then } recd(s) \textit{ else } s|\mathsf{p} \\ manSig(s,\mathsf{r},A,a).\mathsf{p} &:& (manSig(s,\mathsf{r},A,a)|\mathsf{p})\texttt{Type} & := \textit{if } \mathsf{p}{=}\mathsf{r} \textit{ then } A \textit{ else } s.\mathsf{p} \end{array}$$

I have not shown the passing around of proofs of occurrence, but notice that for p and s ground, $occ(\mathsf{p},s)$ computes to `True` or `False`, and the only proof object needed for ground examples is the canonical proof of `True`. Nevertheless, a good deal of computation (which embodies the run-time type checking of the computation rules of Fig. 2) is involved in using this representation.

Computation rules Keeping in mind that the types of records are nested Σ and Ψ (hence "$l.1$" and "$l.2$" in the next definition), we can program the computation rules by $sign$-elimination.

For $l : recd(opSig(s,\mathsf{r},A))$,

$$\begin{array}{rcll} l|\mathsf{p} &:=& \textit{if } \mathsf{p}{=}\mathsf{r} \textit{ then } l.1 \textit{ else } (l.1)|\mathsf{p} & : opSig(s,\mathsf{r},A)|\mathsf{p} \\ l.\mathsf{p} &:=& \textit{if } \mathsf{p}{=}\mathsf{r} \textit{ then } l.2 \textit{ else } (l.1).\mathsf{p} & : opSig(s,\mathsf{r},A).\mathsf{p}(l|\mathsf{p}) \end{array}$$

For $l : recd(manSig(s,\mathsf{r},A,a))$,

$$\begin{array}{rcll} l|\mathsf{p} &:=& \textit{if } \mathsf{p}{=}\mathsf{r} \textit{ then } l.1 \textit{ else } (l.1)|\mathsf{p} & : manSig(s,\mathsf{r},A,a)|\mathsf{p} \\ l.\mathsf{p} &:=& \textit{if } \mathsf{p}{=}\mathsf{r} \textit{ then } a(l.1) \textit{ else } (l.1).\mathsf{p} & : manSig(s,\mathsf{r},A,a).\mathsf{p}(l|\mathsf{p}) \end{array}$$

***with* rules** As explained in Sect. 5.4.1, *with* must be defined simultaneously with a coercion, u, forgetting *with* information. Here is the typing of these operations:

$$\begin{array}{rcl} with &:& (s{:}\, sign;\ \mathsf{p}{:}\, lbl;\ a{:}\,(l{:}s|\mathsf{p})s.\mathsf{p}(l))sign, \\ u &:& (s{:}\, sign;\ \mathsf{p}{:}\, lbl;\ a{:}\,(l{:}s|\mathsf{p})s.\mathsf{p}(l))(recd(s \textit{ with } \mathsf{p}{=}a))recd(s). \end{array}$$

[6] For clarity, I show the first argument to Σ and Ψ, which has been suppressed above.

As above, I have suppressed the required proof of $occ(\mathsf{p}, s)$. In fact, to program exactly the rules of Fig. 8, which do not allow *with* to over-write an already manifest field, requires a new relation saying that the outermost occurrence of p is opaque in s. Here are the definitions:

$$
\begin{array}{rcll}
opSig(s, \mathsf{r}, B) \ with \ \mathsf{p}{=}\, a & := & if \ \mathsf{r}{=}\mathsf{p} & then \ manSig(s, \mathsf{r}, B, a) \\
 & & & else \ opSig(s \ with \ \mathsf{p}{=}\, a, \mathsf{r}, B{\circ}u(s, \mathsf{p}, a)), \\
u(opSig(s, \mathsf{r}, B), \mathsf{p}, a, l) & := & if \ \mathsf{r}{=}\mathsf{p} & then \ \langle l, a(l) \rangle \\
 & & & else \ \langle u(s, \mathsf{p}, a, l.1), l.2 \rangle, \\
manSig(s, \mathsf{r}, B, b) \ with \ \mathsf{p}{=}\, a & := & \multicolumn{2}{l}{manSig(s \ with \ \mathsf{p}{=}\, a, \mathsf{r}, B{\circ}u(s, \mathsf{p}, a), b{\circ}u(s, \mathsf{p}, a)),} \\
u(manSig(s, \mathsf{r}, B, b), \mathsf{p}, a, l) & := & \multicolumn{2}{l}{\langle u(s, \mathsf{p}, a, l.1) \rangle_a.}
\end{array}
$$

It remains to experiment with this encoding, and develop interesting examples. This representation has first class labels with testable equality, which suggests possibilities for programmable renaming and opening.

7 Conclusions

I have shown surprisingly simple first-class dependently typed records with manifest types. The manifest types are extended by a simple *with* notation for adding manifest information to an existing signature. There is no built-in notion of subtyping, but coercive subtyping gives a more flexible notion than the usual structural record subtyping. The *with* notation, which has always been treated as syntactic sugar in the literature, is completely formalized in my presentation.

I have not developed any meta-theory, but these records, including first class labels with testable equality, can be programmed in type theory extended with inductive-recursive definitions. Thus, if the extended framework has good properties, as is informally believed, the system with manifest signatures preserves these properties. These records are suggested for formalizing mathematical structures, not necessarily as modules for separate checking or proof libraries. Further work remains on several important aspects of usability, such as efficiency and renaming of fields.

References

[1] P. Aczel. Simple overloading for type theories. Privately circulated notes, 1994.

[2] A. Bailey. *The Machine-checked Literate Formalisation of Algebra in Type Theory.* PhD thesis, Univ. of Manchester, 1998.

[3] G. Betarte. *Dependent Record Types and Formal Abstract Reasoning.* PhD thesis, Chalmers Univ. of Technology, 1998.

[4] 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 Univ. Press, 1998.

[5] The Coq Project, 1999. `http://pauillac.inria.fr/coq/`.

[6] J. Courant. MC: A module calculus for Pure Type Systems. Technical Report 1217, CNRS Université Paris Sud 8623: LRI, June 1999.

[7] N. G. de Bruijn. Telescopic mappings in typed lambda calculus. *Information and Computation*, 91(2):189–204, April 1991.

[8] Peter Dybjer. A general notion of simultaneous inductive-recursive definition in type theory. *Journal of Symbolic Logic*, 1997. To appear.

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

[10] F. Kammüller. Modular structures as dependent types in isabelle. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *TYPES'98, Selected Papers*, volume 1657 of *LNCS*. Springer-Verlag, 1999.

[11] B. Lampson and R. Burstall. Pebble, a kernel language for modules and abstract data types. *Information and Computation*, 76(2/3), 1988.

[12] The LEGO Proof Assistant, 1999. `http://www.dcs.ed.ac.uk/home/lego/`.

[13] X. Leroy. Manifest types, modules, and separate compilation. In *POPL'94*, New York, NY, USA, 1994. ACM Press.

[14] Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1), 1999.

[15] Z. Luo and S. Soloviev. Dependent coercions. In *Category Theory in Computer Science, CTCS'99*, Electronic Notes in Theoretical Computer Science. Elsevier, 1999.

[16] D. MacQueen. Using dependent types to express modular structure. In *POPL'86*, 1986.

[17] Loic Pottier. Algebra with Coq. See *User contributions* in Coq release [5], 1999.

[18] A. Saibi. Typing algorithm in type theory with inheritance. *POPL'97*, 1997.

[19] A. Tasistro. *Substitution, Record Types and Subtyping in Type Theory, with Applications to the Theory of Programming.* PhD thesis, Chalmers Univ. of Technology, May 1997.