Dependent Intersection: A New Way of Defining Records in Type Theory *

Alexei Kopylov
Department of Computer Science
Cornell University
Ithaca, NY 14853, USA

Abstract

Records and dependent records are a powerful tool for programming, representing mathematical concepts, and program verification. In the last decade several type systems with records as primitive types were proposed. The question is arisen: whether it is possible to define record type in existent type theories using standard types without introducing new primitives.

It was known that independent *records can be defined in type theories with dependent functions or intersection. On the other hand* dependent *records cannot be formed using standard types. Hickey introduced a complex notion of* very dependent functions *to represent dependent records. In the current paper we introduce a simpler type constructor* dependent intersection*, i.e., the intersection of* two *types, where the second type may depend on elements of the first one (do not confuse it with the intersection of a family of types). This new type constructor allows us to define dependent records in a very simple way and also inherently interesting on its own.*

1 Introduction

In general, records are tuples of labeled fields, where each field may have its own type. In dependent records (or more formally dependently typed records) type of components may depend on values of the other components. We will consider type theories where types are first-class objects. In such theories values of record components may be types. This makes the notion of dependent records very powerful. Dependent records may be used to represent algebraic structures (such as groups) and modules in programming languages like SML or Haskell (see for example [4, 10]). Moreover if type theory has propositions-as-types principle [9, 18], then we can add specifications (i.e. properties of the intended behavior) of a module as additional components to this module. [11] suggests to use dependent records for mathematical theories where fields of records are theorems. Records are also the first step toward object-oriented calculi [1].

Example 1 *One can define the signature for ordered set as a dependent record type:*

$$OrdSetSig = \{\mathtt{t} : \mathit{Type}; \mathtt{less} : \mathtt{t} \to \mathtt{t} \to \mathbb{B}\}$$

This definition can be understood as a signature of an algebraic structure as well as an interface of a module in a programing language.

Example 2 *The proposition-as-type principle allows us to add the property of ordered sets as a new component:*

$$OrdSet = \{\mathtt{t} : \mathit{Type}; \mathtt{less} : \mathtt{t} \to \mathtt{t} \to \mathbb{B}; \mathtt{axm} : EqRel(\mathtt{t}, \mathtt{less})\}$$

where $EqRel(\mathtt{t}, \mathtt{less})$ *is a predicate stating that* `less` *is an equivalence relation on* `t`*.*

*This work was supported in part by the DoD Multidisciplinary University Research Initiative (MURI) program administered by the Office of Naval Research (ONR) under Grant N00014-01-1-0765, the Defense Advanced Research Projects Agency (DARPA) under Grant F30602-98-2-0198, and by NSF Grant CCR 0204193.

Here axm is a new field that defines axiom of the algebraic structure of ordered sets (or specification of the module $OrdSet$).

Example 3 *In type theories with equality, manifested fields ([16]) may be also represented as specification.*

$$IntOrdSetSig = \{\mathtt{t} : \mathit{Type}; \mathtt{less} : \mathtt{t} \to \mathtt{t} \to \mathbb{B}; \mathtt{mnf} : \mathtt{t} = \mathbb{Z}\}$$

is a signature where type t *is bound to be type of integers.*

From mathematical point of view the record type is similar to the product type. The essential difference is the subtyping property: we can extend a record type with new fields and get a subtype of the original record type. E.g. $OrdSet$ and $IntOrdSetSig$ defined above are subtypes of $OrdSetSig$. Subtyping property is important in mathematics: we can apply all theorems about monoid to smaller types such as groups. It is also essential in programing for inheritance and abstractions.

Different type theories with records were proposed both for proof systems as well as for programming languages ([10, 16, 8, 4, **?**, 21] and others). These systems treat record type as a new primitive. In the current paper we are interesting in the following natural question: *whether it is possible to express the notion of records in usual type theories without record type as primitives?* This question is especially interesting for pure mathematical proof systems. As we saw records are a handy tool to represent algebraic structures. On the other hand records do not seem to be the basic mathematical concept that should be included in the foundation of mathematics. Records should be rather defined in terms of more abstract mathematical concepts.

It turned out that it is possible to define *independent records* in a sufficient powerful type theory that has dependent functions [11, 6] or intersection [**?**]. On the other hand, there is no known way to form dependent records in standard type theories (cf. [**?**]). However, Hickey [11] showed that *dependent records* can be formed in an extension of Martin-Löf type theory. Namely, he introduced a new type of *very depended functions*. This type is powerful enough to express dependent records in a type theory and provides a solid mathematical foundation of dependent records. Unfortunately the type of very depended functions is very complex itself. The rules and the semantics probably is more complicated for this type than for dependent records. The question is whether there is a simpler way to add dependent records to a type theory?

In this paper we extend a type theory with a simple and easier to understand primitive type constructor, *dependent intersection*. This is a natural generalization of the standard intersection introduced in [**?**] and [22]. Dependent intersection is an intersection of *two* types, where the second type may depend on elements of the first one. This type constructor is built by analogy to dependent products: elements of dependent product are pairs where the type of the second component may depend on the first component. We will show that dependent intersection allows us to define the record type in a very simple way. Our definition of records is extensionally equal to Hickey's ones, but is far simpler. Moreover our constructors (unlike Hickey's) allow us to extend record types. For example, having a definition of monoids we can define groups by extending this definition rather than repeating the axioms of monoid. We also show some other interesting uses of this new type.

The structure of the paper is as follows. In Section 2 we recall the concept of the ordinary intersection, then we introduce a new type, *dependent intersection*, give the semantics for this type and the corresponding inference rules. Then, in Section 3 we informally describe records and their intended properties. We show Hickey's definition of records and show how record types are connected to intersections. In Section 4 we present the formal definition of records as dependent intersection. In Appendix B we give an example of an abstract data type ($Stack$) represented in our type theory as a record. In Section 5 we show another application of dependent intersection: the set type constructor $\{x : T \mid P(x)\}$ can be defined as dependent intersection as well.

The theory of dependent intersection and dependent records is implemented in the MetaPRL system [12, 13]. See theories `itt_disect` and `itt_record*` in Logical Theories in [13]. The MetaPRL system is based on the NuPRL type theory [7], which is a generalization of Martin-Löf's type theory [18].

The NuPRL type theory has proposition-as-types principle and powerful enough to represent any mathematical statement. NuPRL has equality, subtyping relation and intersection type. Membership and subtyping are extensional. For example, $A \subseteq B$ does not say anything about structure of these types, but only means that if $x \in A$ then $x \in B$. As a result the type checking and subtyping are undecidable.

2 Dependent Intersection

2.1 Ordinary Intersection

It is well known that the binary intersection can be added to a type theory. See for example [19].

The intersection of two types A and B is a new type containing elements that are both in A and B. For example, $\lambda x.x+1$ is an element of the type $(\mathbb{Z} \to \mathbb{Z}) \bigcap (\mathbb{N} \to \mathbb{N})$. Two elements are considered to be equal as elements of the type $A \bigcap B$ if they are equal in both types A and B.

Example 4 *Let $A = \mathbb{N} \to \mathbb{N}$ and $B = \mathbb{Z}^- \to \mathbb{Z}$ (where $\mathbb{Z}^-$ is a type of negative integers). Then $id = \lambda x.x$ and $abs = \lambda x.|x|$ are both elements of the type $A \bigcap B$. Although id and abs are equal as elements of the type $\mathbb{N} \to \mathbb{N}$ (because these two functions do not differ on $\mathbb{N}$), id and abs are different as elements of $\mathbb{Z}^- \to \mathbb{Z}$. Therefore, $id \neq abs \in A \bigcap B$.*

2.2 New Intersection

We extend this definition to a case when type B can depend on elements of type A. Let A be a type and $B[x]$ be a type for all x of the type A. We define a new type, *dependent intersection* $x : A \bigcap B[x]$. This type contains all elements a from A such that a is also in $B[a]$.

Remark 5 *Do not confuse* the dependent intersection *with* the intersection of a family of types $\bigcap_{x:A} B[x]$. *The later refers to an intersection of types $B[x]$ for all x in A. The difference between these two type constructors is similar to the difference between dependent products $x : A \times B[x] = \Sigma_{x:A} B[x]$ and the product of a family of types $\Pi_{x:A} B[x] = x : A \to B[x]$.*

Example 6 *The ordinary binary intersection is just a special case of a dependent intersection with a constant second argument*

$$A \bigcap B = x : A \bigcap B.$$

Example 7 *Let $A = \mathbb{Z}$ and $B[x] = \{y : \mathbb{Z} \mid y > 2x\}$ (i.e. $B[x]$ is a type off all integers y, s.t. $y > 2x$). Then $x : A \bigcap B[x]$ is a set of all integers, such that $x > 2x$.*

Two elements a and a' are equal in the dependent intersection $x : A \bigcap B[x]$ when they are equal both in A and $B[a]$.

Example 8 *Let $A = \{0\} \to \mathbb{N}$ and $B[f] = \{1\} \to \mathbb{N}_{f(0)}$, where $\mathbb{N}_n$ is a type of the first n natural numbers and $\{0\}$ and $\{1\}$ are types that contain only one element (0 and 1 correspondingly). Then $x : A \bigcap B[x]$ is a type of functions f that map 0 to a natural number n_0 and map 1 to a natural number $n_1 \in \{0, 1, \ldots, n_0 - 1\}$. Two such functions f and f' are equal in this type, when first, $f = f' \in \{0\} \to \mathbb{N}$, i.e. $f(0) = f'(0)$, and second, $f = f' \in \{1\} \to \mathbb{N}_{f(0)}$, i.e. $f(1) = f'(1) < f(0)$.*

2.3 Semantics

We are going to give the formal semantics for dependent intersection types based on the predicative PER semantics of [2, 3].

We are supposed that our type theory has the judgments of the following forms:

A Type	A is a well-formed type
$A = B$	A and B are (intentionally) equal types
$a \in A$	a has type A
$a = b \in A$	a and b are equal as elements of type A

In the PER semantics types are interpreted as partial equivalence relations (PERs) over terms. Partial equivalence relations are relations that transitive and symmetric, but not necessary reflexive.

According to [3], to give the semantics for a type A we need to determine when this type is well-formed and specify the partial equivalence relation for this type ($a = b \in A$). Then we define $a \in A$ to be true when $a = a \in A$. We should also give an equivalence relation on types, i.e. determine when two types are equal.

The partial equivalence relation $a = b \in A$ may be defined in two steps. First, we say when a term is an element of the given type (i.e. specify when $a \in A$ is true). Second, we define an *equivalence* relation on elements of this type.

The Extension of the Semantics We introduce a new term constructor for dependent intersection $x : A \bigcap B[x]$. This constructor bounds the variable x in B. We extend the semantics of [3] as follows.

- The expression $x : A \bigcap B[x]$ is a well-formed type if and only if A is a type and $B[x]$ is a functional type over $x : A$. That is, for any x from A the expression $B[x]$ should be a type and if $x = x' \in A$ then $B[x] = B[x']$.

$$\frac{\Gamma \vdash A\,\text{Type} \qquad \Gamma; x : A \vdash B[x]\,\text{Type}}{\Gamma \vdash (x : A \bigcap B[x])\,\text{Type}} \quad (TypeFormation)$$

$$\frac{\Gamma \vdash A = A' \qquad \Gamma; x : A \vdash B[x] = B'[x]}{\Gamma \vdash (x : A \bigcap B[x]) = (x : A' \bigcap B'[x])} \quad (TypeEquality)$$

$$\frac{\Gamma \vdash a \in A \qquad \Gamma \vdash a \in B[a] \qquad \Gamma \vdash x : A \bigcap B[x]\,\text{Type}}{\Gamma \vdash a \in (x : A \bigcap B[x])} \quad (Introduction)$$

$$\frac{\Gamma \vdash a = a' \in A \qquad \Gamma \vdash a = a' \in B[a] \qquad \Gamma \vdash x : A \bigcap B[x]\,\text{Type}}{\Gamma \vdash a = a' \in (x : A \bigcap B[x])} \quad (Equality)$$

$$\frac{\Gamma; u : (x : A \bigcap B[x]); \Delta[u]; x : A; y : B[x] \vdash C[x, y]}{\Gamma; u : (x : A \bigcap B[x]); \Delta[u] \vdash C[u; u]} \quad (Elimination)$$

Table 1. Inference rules for dependent intersection

- The elements of the well-formed type $x : A \bigcap B[x]$ are such terms a that a is an element of both types A and $B[a]$.
- Two elements a and a' are equal in the well-formed type $x : A \bigcap B[x]$ iff $a = a' \in A$ and $a = a' \in B[a]$. (Note that, since $a = a' \in A$ implies $B[a] = B[a']$, this definition is symmetric, i.e. it does not matter whether we write $a = a' \in B[a]$ or $a = a' \in B[a']$).
- Two types $x : A \bigcap B[x]$ and $x : A' \bigcap B'[x]$ are equal when A and A' are equal types and for all x and y from A if $x = y \in A$ then $B[x] = B'[y]$.

Theorem 9 *The semantics given above is an consistent extension of the standard semantics.*

This theorem can be proved using the standard technique of [3].

2.4 The Inference Rules

The corresponding inference rules are presented in Table 1.

Note that rules *(TypeFormation)* and *(Introduction)* are redundant when we define $A\,\text{Type} \triangleq (A = A)$ and $a \in A \triangleq (a = a \in A)$.

Theorem 10 *All rules of Table 1 are valid in the semantics given above.*

This theorem is proved by straightforward application of the semantics definition.

Theorem 11 *The following rules can be derived from the primitive rules of Table 1 in a type theory with the appropriate cut rule.*

$$\frac{\Gamma \vdash a = a' \in (x : A \bigcap B[x])}{\Gamma \vdash a = a' \in A} \quad (CaseEquality1)$$

$$\frac{\Gamma \vdash a = a' \in (x : A \bigcap B[x])}{\Gamma \vdash a = a' \in B[a]} \quad (CaseEquality2)$$

This rules was derived in the MetaPRL system (a system based on the NuPRL type theory) and the proof was machine-checked.

Theorem 12 *Dependent intersection is associative, i.e.*

$$x : A \bigcap (y : B[x] \bigcap C[x, y]) =_e z : (x : A \bigcap B[x]) \bigcap C[z, z]$$

where $=_e$ stands for extensional equality, that is $T_1 =_e T_2$ when $T_1 \subseteq T_2$ and $T_2 \subseteq T_1$, i.e. these two types have the same elements and the same equality relations.

The formal proof was also checked by the MetaPRL system. We show here only an informal proof. An element x has type $a : A \bigcap (b : B[a] \bigcap C[a, b])$ iff it has types A and $b : B[x] \bigcap C[x, b]$. The later is a case iff $x \in B[x]$ and $x \in C[x, x]$. On the other hand, x has type $ab : (a : A \bigcap B[a]) \bigcap C[ab, ab]$ iff $x \in (a : A \bigcap B[a])$ and $x \in C[x, x]$. The former means that $x \in A$ and $x \in B[x]$. Therefore $x \in a : A \bigcap (b : B[a] \bigcap C[a, b])$ iff $x \in A$ and $x \in B[x]$ and $x \in C[x, x]$ iff $x \in ab : (a : A \bigcap B[a]) \bigcap C[ab, ab]$.

3 Records

We are going to define record type using dependent intersection. In this section we informally describe what properties we are expecting from records. The formal definitions are presented in Section 4.

3.1 Plain Records

Records are collection of labeled fields. We use the following notations for records:

$$\{\mathtt{x}_1 = a_1, \ldots, \mathtt{x}_n = a_n\} \tag{1}$$

where $\mathtt{x}_1, \ldots, \mathtt{x}_n$ are *labels* and $a_1, \ldots a_n$ are corresponding fields. Usually labels have a string type, but generally speaking labels can be of any fixed type $Label$ with a decidable equality. We will use `the truetype font` for labels.

The extraction operator $r.\mathtt{x}$ is used to access record fields. If r is a record then $r.\mathtt{x}$ is a field of this record labeled $\mathtt{x}$. That is we expect the following reduction rule:

$$\{\mathtt{x}_1 = a_1, \ldots, \mathtt{x}_n = a_n\}.\mathtt{x}_i \longrightarrow a_i.$$

Fields may have different types. If each a_i has type A_i then the whole record (1) has the type

$$\{\mathtt{x}_1 : A_1, \ldots, \mathtt{x}_n : A_n\}. \tag{2}$$

Also we want the natural typing rule for the field extraction: for any record r of the type (2) we should be able to conclude that $r.\mathtt{x}_i \in A_i$.

The main difference between record types and products $A_1 \times \cdots \times A_n$ is that record type has the *subtyping property*. Given two records R_1 and R_2, if any label declared in R_1 as a field of type A is also declared in R_2 as a field of type B, such that $B \subseteq A$, then R_2 is subtype of R_1. In particular,

$$\{\mathtt{x}_1 : A_1, \ldots, \mathtt{x}_n : A_n\} \subseteq \{\mathtt{x}_1 : A_1, \ldots, \mathtt{x}_m : A_m\} \tag{3}$$

where $m < n$.

Example 13 *Let $Point = \{\mathtt{x} : \mathbb{Z}; \mathtt{y} : \mathbb{Z}\}$ and $ColorPoint = \{\mathtt{x} : \mathbb{Z}; \mathtt{y} : \mathbb{Z}; \mathtt{color} : Color\}$. Then the record $\{\mathtt{x} = 0; \mathtt{y} = 0; \mathtt{color} = red\}$ is not only a $ColorPoint$, but it is also a $Point$, so we can use this record whenever $Point$ is expected. For example, we can use it as an argument of the function of the type $Point \to T$. Further the result of this function does not depend whether we use $\{\mathtt{x} = 0; \mathtt{y} = 0; \mathtt{color} = red\}$ or $\{\mathtt{x} = 0; \mathtt{y} = 0; \mathtt{color} = green\}$. That is, these two records are equal as elements of the type $Point$, i.e.*

$$\begin{aligned}&\{\mathtt{x} = 0; \mathtt{y} = 0; \mathtt{color} = red\} = \\ &\{\mathtt{x} = 0; \mathtt{y} = 0; \mathtt{color} = green\} \in \{\mathtt{x} : \mathbb{Z}; \mathtt{y} : \mathbb{Z}\}\end{aligned}$$

This is a natural corollary from the subtyping property.

Using subtyping one can module the abstract fields. Consider a record r that has one "private" field $\mathtt{x}$ of the type A and one "public" field $\mathtt{y}$ of the type B. This record has type $\{\mathtt{x} : A; \mathtt{y} : B\}$ Using subtyping property we can conclude that it also has type $\{\mathtt{y} : B\}$. Now we can consider type $\{\mathtt{y} : B\}$ as a public interface for this record. Anyone who knows that $r \in \{\mathtt{y} : B\}$ have access to field $\mathtt{y}$, but access to field $\mathtt{x}$ would be type invalid. That is, a function of the type $\{\mathtt{y} : B\} \to T$ can manipulate only with field $\mathtt{y}$ on its argument, but we can pass a record with field $\mathtt{x}$ to it.

Further, records do not depend on field ordering. For example, $\{\mathtt{x} = 0; \mathtt{y} = 1\}$ should be equal to $\{\mathtt{y} = 1; \mathtt{x} = 0\}$, moreover $\{\mathtt{x} : A; \mathtt{y} : B\}$ and $\{\mathtt{y} : B; \mathtt{x} : A\}$ should define the same type.

3.1.1 Records as Dependent Functions

Records may be considered as mappings from labels to the corresponding fields. Therefore it is natural to define a record type as a function type with the domain $Label$ (cf. [6]). Since the types of each field may vary, one should use dependent function type (i.e., Π type). Let $Field[l]$ be a type of a field labeled l. For example, for the record type (2) take

$$\begin{aligned} Field[l] \triangleq \; & \\ & \mathtt{if}\ l = \mathtt{x}_1\ \mathtt{then}\ A_1\ \mathtt{else} \\ & \ldots \\ & \mathtt{if}\ l = \mathtt{x}_n\ \mathtt{then}\ A_n \\ & \ \mathtt{else}\ \mathrm{Top} \end{aligned}$$

Then define the record type as the dependent function type:[1]

$$\{\mathtt{x}_1 : A_1; \ldots; \mathtt{x}_n : A_n\} \triangleq l : Label \to Field[l]. \tag{4}$$

Now records may be defined as functions:

$$\begin{aligned} \{\mathtt{x}_1 = a_1; \ldots; \mathtt{x}_n = a_n\} \triangleq \; & \\ & \lambda l.\mathtt{if}\ l = \mathtt{x}_1\ \mathtt{then}\ a_1\ \mathtt{else} \\ & \quad \ldots \\ & \quad \mathtt{if}\ l = \mathtt{x}_n\ \mathtt{then}\ a_n \end{aligned} \tag{5}$$

And extraction is defined as application:

$$r.l \triangleq r\, l \tag{6}$$

One can see that these definitions meet the expecting properties mentioned above including subtyping property.

3.1.2 Records as Intersections

Using above definitions we can prove that in case when all $\mathtt{x}_i$'s are distinct labels

$$\{\mathtt{x}_1 : A_1; \ldots; \mathtt{x}_n : A_n\} =_e \{\mathtt{x}_1 : A_1\} \bigcap \ldots \bigcap \{\mathtt{x}_n : A_n\}. \tag{7}$$

This property provides us a simpler way to define records. First, let us define the type of records with only one field. We define it as a function type like we did it in the last section, but for single-field records we do not need depend functions, so we may simplify the definition:

$$\{\mathtt{x} : A\} \triangleq \{\mathtt{x}\} \to A \tag{8}$$

where $\{\mathtt{x}\}$ is the singleton subset of $Label$.

Now we may take (7) as a definition of an arbitrary record type (cf. [**?**]).

Example 14 *The record* $\{\mathtt{x} = 1; \mathtt{y} = 2\}$ *by definition (5) is a function that maps* $\mathtt{x}$ *to* 1 *and* $\mathtt{y}$ *to* 2. *Therefore it has type* $\{\mathtt{x}\} \to \mathbb{Z} = \{\mathtt{x} : \mathbb{Z}\}$ *and also has type* $\{\mathtt{y}\} \to \mathbb{Z} = \{\mathtt{y} : \mathbb{Z}\}$. *Hence it has type* $\{\mathtt{x} : \mathbb{Z}; \mathtt{y} : \mathbb{Z}\} = \{\mathtt{x} : \mathbb{Z}\} \bigcap \{\mathtt{y} : \mathbb{Z}\}$.

[1] We use the standard NuPRL notations $x : A \to B[x] = \prod_{x:A} B[x]$ for the type of functions that maps each $x \in A$ to an element of the type $B[x]$.

One can see that when all labels are distinct definitions (4) and (7)+(8) are equivalent. That is, for any record expression $\{x_1 : A_1; \ldots; x_n : A_n\}$ where $x_i \neq x_j$, these two definitions define two extensionally equal types.

However, definitions (7)+(8) differ from the traditional ones, in the case when labels may coincide. Most record calculi prohibit repeating labels in the declaration of record types, e.g., they do not recognize the expression $\{\mathtt{x} : A; \mathtt{x} : B\}$ as a valid type. On the other hand, in [11] in the case when label coincide the last field overlap the previous ones, e.g., $\{\mathtt{x} : A; \mathtt{x} : B\}$ is equal to $\{\mathtt{x} : B\}$. In both these cases many typing rules of the record calculus need some additional conditions that prohibits coincident labels. For example, the subtyping relation (3) would be true only when all labels $\mathtt{x}_i$ are distinct.

We will follow the definition (7) and allow repeated labels and assume that

$$\{\mathtt{x} : A; \mathtt{x} : B\} = \{\mathtt{x} : A \bigcap B\}. \tag{9}$$

This may look unusual, but this notation significantly simplifies the rules of the record calculus, because we do not need to worry about coincident labels. Moreover, this allow us to have multiply inheriting (see Section 4.3 for an example). Note that the equation (9) holds also in Bickford's definition of records [**?**].

3.2 Dependent Records

We want be able to represent abstract data types and algebraic structures as records. For example, a semigroup may be considered as a record with the fields `car` (representing a carrier) and `product` (representing a binary operation). The type of `car` is the universe $\mathbb{U}_i$ (the type of types). The type of `product` should be $\mathtt{car} \times \mathtt{car} \to \mathtt{car}$. The problem is that the type of `product` depends on the value of the field `car`. Therefore we cannot use plain record types to represent such structures.

We need dependent records [**?**, 11, 21]. In general a dependent record type has the following form

$$DepRec = \{\mathtt{x} : A; \mathtt{y} : B[\mathtt{x}]; \mathtt{z} : C[\mathtt{x}, \mathtt{y}]; \ldots\}$$

That is, the type of a field in such records can depend on the values of the previous fields.

The following two properties show the intended meaning of this type.

1. The record $\{\mathtt{x} = a; \mathtt{y} = b; \mathtt{z} = c; \ldots\}$ has type $DepRec$ when

$$a \in A, \quad b \in B[a], \quad c \in C[a, b], \quad \ldots$$

2. For any record $r \in DepRec$

$$r.\mathtt{x} \in A, \quad r.\mathtt{y} \in B[r.\mathtt{x}], \quad r.\mathtt{z} \in C[r.\mathtt{x}, r.\mathtt{y}], \quad \ldots$$

Example 15 *Let $SemigroupSig$ be the record type that represents the signature of semigroups:*

$$SemigroupSig = \{\mathtt{car} : \mathbb{U}_i; \mathtt{product} : \mathtt{car} \times \mathtt{car} \to \mathtt{car}\}.$$

Semigroups are elements of $SemigroupSig$ satisfying the associative axiom. This axiom may be represented as an additional field.

$$\begin{aligned} Semigroup = \{\ & \mathtt{car} : \mathbb{U}_i; \\ & \mathtt{product} : \mathtt{car} \times \mathtt{car} \to \mathtt{car}; \\ & \mathtt{axm} : \forall x, y, z : \mathtt{car}.\ (x \cdot y) \cdot z = x \cdot (y \cdot z)\} \end{aligned}$$

where $x \cdot y$ stands for $\mathtt{product}(x, y)$.

3.2.1 Dependent Records as Very Dependent Functions

We cannot define dependent record type using "ordinary" dependent function type, because the type of the fields depends not only on labels, but also on values of other fields.

To represent dependent records Hickey [11] introduced the *very dependent function* type constructor:

$$\{f \mid x : A \to B[f, x]\} \tag{10}$$

Here A is the domain of the function type and the range $B[f, x]$ can depends on the argument x and the function f itself. That is type (10) refers to the type of all functions g with the domain A and the range $B[g, a]$ on any argument $a \in A$.

For instance, $SemigroupSig$ can be represented as a very dependent function type

$$SemigroupSig \triangleq \{r \mid l : Label \to Field[r, l]\} \tag{11}$$

where $Field[r, l] \triangleq$

$$\begin{array}{l} \texttt{if } l = \texttt{car then } \mathbb{U}_i \texttt{ else} \\ \texttt{if } l = \texttt{product then } r.\texttt{car} \times r.\texttt{car} \to r.\texttt{car} \\ \;\texttt{else Top} \end{array}$$

Not every very dependent function type has a meaning. For example the range of the function on argument a cannot depend on $f(a)$ itself. For instance, the expression

$$\{f \mid x : A \to f(x)\}$$

is not a well-formed type.

The type (10) is well-formed if there is some well-founded order $<$ on the domain A, and the range type $B[x, f]$ on $x = a$ depends only on values $f(b)$, where $b < a$. The requirement of well-founded order makes the definition of very-dependent functions to be very complex. See [11] for more details.

3.2.2 Dependent Records as Dependent Intersection

By using dependent intersection we can avoid the complex concept of very dependent functions. For example, we may define

$$\begin{array}{l} SemigroupSig \triangleq self : \{\texttt{car} : \mathbb{U}_i\} \bigcap \\ \{\texttt{product} : self.\texttt{car} \times self.\texttt{car} \to self.\texttt{car}\} \end{array}$$

Here $self$ is a bound variable that is used to refer to the record itself considered as a record of the type $\{\texttt{car} : \mathbb{U}_i\}$. This definition can be read as following:

> r has type $SemigroupSig$, when first, r is a record with a field car of the type $\mathbb{U}_i$, and second, r is a record with a field product of the type $r.\texttt{car} \times r.\texttt{car} \to r.\texttt{car}$.

This definition of the $SemigroupSig$ type is equal to (11), but it has two advantages. First, it is much simpler. Second, dependent intersection allows us to extend the $SemigroupSig$ type to the $Semigroup$ type by adding an extra field axm:

$$\begin{array}{l} Semigroup \triangleq self : SemigroupSig \bigcap \\ \{\texttt{axm} : \forall x, y, z : self.\texttt{car} \quad (x \cdot y) \cdot z = x \cdot (y \cdot z)\} \end{array}$$

where $x \cdot y$ stands for $self.\texttt{product}(x, y)$.

We can define dependent record type of an arbitrary length in this fashion as a dependent intersection of single-field records associated to the left.

Note that $Semigroup$ can be also defined as an intersection associated to the right: $Semigroup =$

$$\begin{array}{ll} r_c : & \{\texttt{car} : \mathbb{U}_i\} \bigcap \\ \big(r_p : & \{\texttt{product} : r_c.\texttt{car} \times r_c.\texttt{car} \to r_c.\texttt{car}\} \bigcap \\ & \{\texttt{axm} : \forall x, y, z : r_c.\texttt{car} \quad (x \cdot y) \cdot z = x \cdot (y \cdot z)\}\big) \end{array}$$

where $x \cdot y$ stands for $r_p.\texttt{product}(x, y)$. Here r_c and r_p are bound variables. Both of them refer to the record itself, but r_c has type $\{\texttt{car} : \mathbb{U}_i\}$ and r_p has type $\{\texttt{product} : \ldots\}$. These two definitions are equal, because of associativity of dependent intersection (Theorem 12).

Note that Robert Pollack [21] considered two types of depended records: left associating records and right associating records. However, in our framework left and right association are just two different ways of building the same type. We will allow using both of them. Which one to chose is the matter of taste.

4 Record Calculus

4.1 The Formal Definitions

Now we are going to give the formal definitions of records using dependent intersection.

4.1.1 Records

Elements of record types are defined as previous. They map labels to the corresponding fields. We could pick any function as a definition of an empty record:

$$\{\} \triangleq \lambda l.l$$

The definitions of field update/extension and field selection are the same as in [11]:

$$(r.\mathtt{x} := a) \triangleq (\lambda l.\mathtt{if}\ l = \mathtt{x}\ \mathtt{then}\ a\ \mathtt{else}\ r\ l)$$
$$r.\mathtt{x} \triangleq r\ x$$

These are basic operations. We can construct any record by these operations. That is, we will consider record $\{\mathtt{x}_1 = a_1; \ldots; \mathtt{x}_n = a_n\}$ as a syntax sugar for expression:

$$\{\}.\mathtt{x}_1 := a_1.\mathtt{x}_2 := a_2.\ldots.\mathtt{x}_n := a_n$$

These definitions provides that

$$\begin{aligned}\{\mathtt{x}_1 = a_1; \ldots; \mathtt{x}_n = a_n\} = \lambda l.&\mathtt{if}\ l = \mathtt{x}_1\ \mathtt{then}\ a_1\ \mathtt{else}\\ &\ldots\\ &\mathtt{if}\ l = \mathtt{x}_n\ \mathtt{then}\ a_n\end{aligned}$$

4.1.2 Record Types

Single-field record type is defined as

$$\{\mathtt{x} : A\} \triangleq \{\mathtt{x}\} \to A$$

where $\{\mathtt{x}\} \triangleq \{l : Label \mid l = \mathtt{x} \in Label\}$ is a singleton set[2].

Independent concatenation of record types is defined as

$$\{R_1; R_2\} \triangleq R_1 \bigcap R_2$$

Dependent concatenation of record type (left associative) is defined as

$$\{self : R_1; R_2[self]\} \triangleq self : R_1 \bigcap R_2[self]$$

Syntactical Remarks Here $self$ is a variable bounded in R_2. We will usually use the name "self" for this variable and use the shortening $\{R_1; R_2[self]\}$ for this type. Further, we will omit "$self$." in the body of R_2, e.g. we will write just $\mathtt{x}$ for $self.\mathtt{x}$, when such notation does not lead to misunderstanding.

We assume that this concatenation is a left associative operation and we will omit inner braces. For example, we will write $\{\mathtt{x} : A; \mathtt{y} : B[self]; \mathtt{z} : C[self]\}$ instead of $\{\{\{\mathtt{x} : A\}; \{\mathtt{y} : B[self]\}\}; \{\mathtt{z} : C[self]\}\}$. Note that in this expression there are two distinct bound variable $self$. First one is bound in B and refers to the record itself as a record of the type $\{\mathtt{x} : A\}$. Second $self$ is bound in C, it also refers to the same record, but it has type $\{\mathtt{x} : A; \mathtt{y} : B[self]\}$.

[2] $\{x : A \mid P[x]\}$ is a standard type constructor in the NuPRL type theory [7]. See also Section 5.

Note that the definition of independent concatenation is just a partial case of dependent concatenation, when R_2 does not depend on $self$.

The definition of right associating records is

$$\{x : \mathtt{x} : A; R[x]\} \triangleq \mathit{self} : \{\mathtt{x} : A\} \bigcap R[\mathit{self}.\mathtt{x}]$$

Syntactical Remarks Here x is a variable bounded in R that represents a field x. Note that we can α-convert the variable x, but not a label x, e.g., $\{x : \mathtt{x} : A; R[x]\} = \{y : \mathtt{x} : A; R[y]\}$, but $\{x : \mathtt{x} : A; R[x]\} \neq \{y : \mathtt{y} : A; R[y]\}$. We will usually use the same name for labels and corresponding bound variables.
This connection is right associative, e.g., $\{x : \mathtt{x} : A; y : \mathtt{y} : B[x]; \mathtt{z} : C[x, y]\}$ stands for $\{x : \mathtt{x} : A; \{y : \mathtt{y} : B[x]; \{\mathtt{z} : C[x, y]\}\}\}$.

4.2 The Rules

The rules of our record calculus are represented in Appendix A. All these rules are derivable from the definitions given above. They were derived in the MetaPRL system.

Then using these rules we can prove the following subtyping properties:

$$\{R_1; R_2\} \subseteq R_1$$
$$\{R_1; R_2\} \subseteq R_2$$
$$\{R_1; R_2[\mathit{self}]\} \subseteq R_1$$
$$\{x : \mathtt{x} : A; R[x]\} \subseteq \{\mathtt{x} : A\}$$

$$\frac{\vdash R_1 \subseteq R_1' \qquad \mathit{self} : R_1 \vdash R_2[\mathit{self}] \subseteq R_2'[\mathit{self}]}{\vdash \{R_1; R_2[\mathit{self}]\} \subseteq \{R_1'; R_2'[\mathit{self}]\}}$$

$$\frac{\vdash A \subseteq A' \qquad x : A \vdash R[x] \subseteq R'[x]}{\vdash \{x : \mathtt{x} : A; R[x]\} \subseteq \{x : \mathtt{x} : A'; R'[x]\}}$$

Further, we can establish two facts that states the equality of left and right associating records.
$\{x : \mathtt{x} : A; R[x]\} =_e \{\mathtt{x} : A; R[\mathit{self}.\mathtt{x}]\}$
$\{R_1; \{x : \mathtt{x} : A[\mathit{self}]; R_2[\mathit{self}, x]\}\} =_e$
$\{\{R_1; \mathtt{x} : A[\mathit{self}]\}; R_2[\mathit{self}, \mathit{self}.\mathtt{x}]\}$
For example, using these two equalities we can prove that
$\{\mathtt{x} : A; \mathtt{y} : B[\mathit{self}.\mathtt{x}]; \mathtt{z} : C[\mathit{self}.\mathtt{x}; \mathit{self}.\mathtt{y}]\} =$
$\{x : \mathtt{x} : A; y : \mathtt{y} : B[x]; \mathtt{z} : C[x; y]\}$
Personally I prefer the left associating records, because first, the left associating constructor allows us to extend records. Second, it is more easily to prove the inclusion of such extension. On the other hand, one can find that right associating records are more natural, because in these records bound variables refer to fields rather than to records itself. It makes dependent records to be more similar to dependent products like $x : A \times (y : B[x] \times C[x, y])$.

4.3 Examples

Semigroup Example Now we can define the $SemigroupSig$ type in two ways:

$$\{\mathtt{car} : \mathbb{U}_i; \mathtt{product} : \mathtt{car} \times \mathtt{car} \to \mathtt{car}\} \quad \text{or}$$
$$\{car : \mathtt{car} : \mathbb{U}_i; \mathtt{product} : car \times car \to car\}$$

Note that in the first definition car in the declaration of product stands for self.car, and in the second definition car is just a bound variable.

We can define $Semigroup$ be extending $SemigroupSig$:

$$\{SemigroupSig; \mathtt{axm} : \forall x, y, z : \mathtt{car} \quad (x \cdot y) \cdot z = x \cdot (y \cdot z)\}$$

or as a right associating record:

$$\begin{aligned}&\{car : \texttt{car} : \mathbb{U}_i;\\ &\ product : \texttt{product} : car \times car \to car;\\ &\ \texttt{axm} : \forall x, y, z : car \quad (x \cdot y) \cdot z = x \cdot (y \cdot z)\}\end{aligned}$$

In the first case $x \cdot y$ stands for $self.\texttt{product}(x, y)$ and in the second case for just $product(x, y)$.

Multiply Inheriting Example A monoid is a semigroup with a unit. So,

$$MonoidSig = \{SemigroupSig; \texttt{unit} : \texttt{car}\}$$

A monoid is an element of $MonoidSig$ which satisfies the axiom of semigroups and an additional property of the unit. That is, $Monoid$ inherits fields from both $MonoidSig$ and $Semigroup$. We can define the $Monoid$ type as follows:

$$\begin{aligned}Monoid = \{\ &MonoidSig; Semigroup;\\ &\texttt{unit\_axm} : \forall x : \texttt{car} \quad x \cdot \texttt{unit} = x\}\end{aligned}$$

Note, that since $MonoidSig$ and $Semigroup$ shared fields car and product, these two fields present in the definition of $Monoid$ twice. This does not create problems, since we allow repeating labels (Section 3.1.2).

Now we have the following subtyping relations:

$$\begin{array}{ccc} SemigroupSig & \supset & MonoidSig \\ \cup & & \cup \\ Semigroup & \supset & Monoid \end{array}$$

5 Sets and Dependent Intersections

By definition, the set type $\{x : T \mid P[x]\}$ is a subtype of T, which contains only such elements x of T that satisfy property $P[x]$ (see [7]). Set types is used to hide a witness of $P[x]$.

Example 16 *The type of natural numbers is defined as* $\mathbb{N} = \{n : \mathbb{Z} \mid n \geq 0\}$. *Without set types we would have to define* $\mathbb{N}$ *as* $n : \mathbb{Z} \times (n \geq 0)$. *In this case we would not have the subtyping property* $\mathbb{N} \subseteq \mathbb{Z}$.

Example 17 *Instead of defining semigroups as records of the* $SemigroupSig$ *type with an additional field* axm, *we could define the* $Semigroup$ *type as a subset of* $SemigroupSig$*:*

$$Semigroup \triangleq \{S : SemigroupSig \mid \forall x, y, z : S.\texttt{car} \ldots\ldots\} \tag{12}$$

In the NuPRL type theory the set type is a primitive type. We will show that the set type may be defined as a dependent intersection.

First, we assume that our type theory has the Top type, that is a supertype of any other type. We will need only one property of the Top type: $T \bigcap Top = T$ for any type T. (In NuPRL Top is defined as $\bigcap_{x:Void} Void$, where $Void$ is the empty type).

Now consider the following type (squash operator):

$$[\![P]\!] \triangleq \{x : \text{Top} \mid P\}$$

$[\![P]\!]$ is an empty type when P is false, and is equal to Top when P is true. The similar type was considered in [15] as a primitive type. We can prove that

$$\{x : T \mid P[x]\} =_e x : T \bigcap [\![P[x]]\!] \tag{13}$$

We could take (13) as a definition of sets, but it would create a loop, because we defined squash operator as a set. To break this loop one can either take squash operator as a primitive type in the way it was done in [15]. That makes sense, because the squash operator is simpler than the set type constructor. Another way to break the loop is to define squash using other primitives. For example, one can define the squash type using union:

$$[P] \triangleq \bigcup_{x:P} \text{Top}.$$

(Union is a type that dual to intersection [19, 12]). In the presence of Markov's principle [15] there is an alternative way to define $[P]$:

$$[P] \triangleq ((A \equiv> \text{Void}) \equiv> \text{Void})$$

where $A \equiv> B \triangleq \bigcap_{x:A} B$.

The Mystery of Notations It is very surprising that braces $\{\dots\}$ was used for sets and for records independently for a long time. But now it turns out that sets and records are almost the same thing, namely, dependent intersection! Compare the definitions for sets and records:

$$\begin{array}{rcl} \{x : T| \quad P[x]\} & \triangleq & x : T \quad \bigcap [P[x]] \\ \{self : R_1; \quad R_2[self]\} & \triangleq & self : R_1 \quad \bigcap R_2[self] \end{array}$$

The only differences between them are that we use squash in the first definition and write "|" for sets and ";" for records.

So, we will use the following definitions for records:

$\{self : R_1 \mid R_2[self]\} \triangleq \{self : R_1; [R_2[self]]\} = self : R_1 \bigcap [R_2[self]]$

$\{x : \mathtt{x} : A \mid R[x]\} \triangleq \{x : \mathtt{x} : A; [R[x]]\} =$
$self : \{\mathtt{x} : A\} \bigcap [R[self.\mathtt{x}]]$

This gives us right to use the shortening notations as in Section 4.1.2 to omit inner braces and "$self$". For example, we can rewrite the definition of the $Semigroup$ type (12) as

$$\begin{array}{rl} Semigroup \triangleq & \{\mathtt{car} : \mathbb{U}_i; \\ & \quad \mathtt{product} : \mathtt{car} \times \mathtt{car} \to \mathtt{car} \mid \\ & \quad \forall x, y, z : \mathtt{car} \quad (x \cdot y) \cdot z = x \cdot (y \cdot z)\} \end{array}$$

Remark Note that we cannot define dependent intersection as a set:

$$x : A \bigcap B[x] \triangleq \{x : A \mid x \in B[x]\}. \qquad \text{(wrong!)}$$

First of all, this set is not well-formed in the NuPRL type theory (this set would be a well-formed type, only when $x \in B[x]$ is a type for all $x \in A$, but the membership is a well-formed type in the NuPRL type theory, only when it is true). Second, this set type does not have the expected equivalence relation. Two elements are equal in this set type, when they are equal just in A, but to be equal in the intersection they must be equal in both types A and B (see Example 4).

6 Acknowledgments

I am very grateful for the productive discussions and useful suggestions to Robert Constable and Aleksey Nogin.

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[2] Stuart F. Allen. A Non-type-theoretic Definition of Martin-Löf's Types. In D. Gries, editor, *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, pages 215–224. IEEE Computer Society Press, June 1987.

[3] Stuart F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987.

[4] Lennart Augustsson. Cayenne — a language with dependent types. In *International Conference on Functional Programming*, pages 239–250, 1998.

[5] Mark Bickford and Jason J. Hickey. Predicate transformers for infinite-state automata in NuPRL type theory. In *Proceedings of* 3^{rd} *Irish Workshop in Formal Methods*, 1999.

[6] Robert L. Constable. Types in logic, mathematics and programming. In Sam Buss, editor, *Handbook of Proof Theory*, chapter 10. Elsevier Science, 1997.

[7] Robert L. Constable et al. *Implementing Mathematics with the NuPRL Development System*. Prentice-Hall, 1986.

[8] Judicaël Courant. An applicative module calculus. In *TAPSOFT*, Lectures Notes in Computer Science, pages 622–636, Lille, France, April 1997. Springer-Verlag.

[9] N. G. deBruijn. The mathematical language Automath: its usage and some of its extensions. In J. P. Seldin and J. R. Hindley, editors, *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61. Springer-Verlag, 1970.

[10] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Conference record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 123–137, Portland, OR, January 1994.

[11] Jason J. Hickey. Formal objects in type theory using very dependent types. In *Foundations of Object Oriented Languages 3*, 1996. Available electronically through the FOOL 3 home page.

[12] Jason J. Hickey. *The MetaPRL Logical Programming Environment*. PhD thesis, Cornell University, Ithaca, NY, January 2001.

[13] Jason J. Hickey, Aleksey Nogin, Alexei Kopylov, et al. MetaPRL home page. `http://metaprl.org/`.

[14] T. B. Knoblock and R. L. Constable. Formalized metareasoning in type theory. In *Proceedings of the 1st Symposium on Logic in Computing Science*, pages 237–248. IEEE, 1986.

[15] Alexei Kopylov and Aleksey Nogin. Markov's principle for propositional type theory. In L. Fribourg, editor, *Computer Science Logic, Proceedings of the* 10^{th} *Annual Conference of the EACSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 570–584. Springer-Verlag, 2001.

[16] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 109–122. ACM Press, 1994.

[17] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[18] Per Martin-Löf. *Intuitionistic Type Theory, Studies in Proof Theory, Lecture Notes*. Bibliopolis, Napoli, 1984.

[19] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, February 1991.

[20] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, February 1991.

[21] Robert Pollack. Dependently typed records for representing mathematical structure. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics:* 13^{th} *International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 461–478. Springer-Verlag, 2000.

[22] G. Pottinger. A type assignment for the strongly normalizable λ-terms. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 561–577. Academic Press, London, 1980.

Reduction rules

$(r.\mathtt{x} := a).\mathtt{x} \longrightarrow a$

$(r.\mathtt{y} := b).\mathtt{x} \longrightarrow r.\mathtt{x}$ when $\mathtt{x} \neq \mathtt{y}$.

In particular: $\{\mathtt{x}_1 = a_1; \ldots; \mathtt{x}_n = a_n\}.\mathtt{x}_i \longrightarrow a_i$ when all $\mathtt{x}_i$'s are distinct.

Type formation

Single-field record:

$$\frac{\Gamma \vdash A \text{ Type} \quad \Gamma \vdash \mathtt{x} \in \mathit{Label}}{\Gamma \vdash \{\mathtt{x} : A\} \text{ Type}}$$

Independent record:

$$\frac{\Gamma \vdash R_1 \text{ Type} \quad \Gamma \vdash R_2 \text{ Type}}{\Gamma \vdash \{R_1; R_2\} \text{ Type}}$$

Dependent record:

$$\frac{\Gamma \vdash R_1 \text{ Type} \qquad \Gamma; \mathit{self} : R_1 \vdash R_2[\mathit{self}] \text{ Type}}{\Gamma \vdash \{R_1; R_2[\mathit{self}]\} \text{ Type}}$$

Right associating record:

$$\frac{\Gamma \vdash \{\mathtt{x} : A\} \text{ Type} \qquad \Gamma; x : A \vdash R[x] \text{ Type}}{\Gamma \vdash \{x : \mathtt{x} : A; R[x]\} \text{ Type}}$$

Introduction (membership rules)

Single-field record:

$$\frac{\Gamma \vdash a \in A \qquad \Gamma \vdash \mathtt{x} \in \mathit{Label}}{\Gamma \vdash r.\mathtt{x} := a \in \{\mathtt{x} : A\}} \qquad \frac{\Gamma \vdash r \in \{\mathtt{x} : A\} \qquad \Gamma \vdash \mathtt{x} \neq \mathtt{y} \in \mathit{Label}}{\Gamma \vdash (r.\mathtt{y} := b) = r \in \{\mathtt{x} : A\}}$$

Independent record:

$$\frac{\Gamma \vdash r \in R_1 \qquad \Gamma \vdash r \in R_2}{\Gamma \vdash r \in \{R_1; R_2\}}$$

Dependent record:

$$\frac{\Gamma \vdash r \in R_1 \qquad \Gamma \vdash r \in R_2[r] \qquad \Gamma \vdash \{R_1; R_2[\mathit{self}]\} \text{ Type}}{\Gamma \vdash r \in \{R_1; R_2[\mathit{self}]\}}$$

Right associating record:

$$\frac{\Gamma \vdash r \in \{\mathtt{x} : A\} \qquad \Gamma \vdash r \in R[r.\mathtt{x}] \qquad \Gamma \vdash \{x : \mathtt{x} : A; R[x]\} \text{ Type}}{\Gamma \vdash r \in \{x : \mathtt{x} : A; R[x]\}}$$

Elimination (inverse typing rules)

Single-field record:

$$\frac{\Gamma \vdash r \in \{\mathtt{x} : A\}}{\Gamma \vdash r.\mathtt{x} \in A}$$

Independent record:

$$\frac{\Gamma \vdash r \in \{R_1; R_2\}}{\Gamma \vdash r \in R_1 \qquad \Gamma \vdash r \in R_2}$$

Dependent record:

$$\frac{\Gamma \vdash r \in \{R_1; R_2[\mathit{self}]\}}{\Gamma \vdash r \in R_1 \qquad \Gamma \vdash r \in R_2[r]}$$

Right associating record:

$$\frac{\Gamma \vdash r \in \{x : \mathtt{x} : A; R[x]\}}{\Gamma \vdash r.\mathtt{x} \in A \qquad \Gamma \vdash r \in R[r.\mathtt{x}]}$$

Table 2. Inference rules for records

A Inference Rules for Record Calculus

The basic rules of our record calculus are shown in Table 2.

We do not show the equality rules here, because in fact, these rules repeat rules in Table 2 and can be derived from them using substitution rules in the NuPRL type theory. For example, we have the following rules

$$\frac{\Gamma \vdash a = a' \in A \qquad \Gamma \vdash \mathtt{x} = \mathtt{x}' \in \mathit{Label}}{\Gamma \vdash (r.\mathtt{x} := a) = (r'.\mathtt{x}' := a') \in \{\mathtt{x} : A\}}$$

$$\frac{\Gamma \vdash r = r' \in R_1 \qquad \Gamma \vdash r = r' \in R_2}{\Gamma \vdash r = r' \in \{R_1; R_2\}}$$

In particular, we can prove that

$$\{\mathtt{x} = 0; \mathtt{y} = 0; \mathtt{color} = \mathit{red}\} = \\ \{\mathtt{x} = 0; \mathtt{y} = 0; \mathtt{color} = \mathit{green}\} \in \{\mathtt{x} : \mathbb{Z}; \mathtt{y} : \mathbb{Z}\}$$

B Final Example: Abstract Data Type

We can represent abstract data types as dependent records. For example, we can define data structure stack as following:

$$\begin{array}{l} \mathit{Stack}(A) \triangleq \\ \{\mathtt{car} : \mathbb{U}_i; \\ \ \mathtt{empty} : \mathtt{car}; \\ \ \mathtt{push} : \mathtt{car} \to A \to \mathtt{car}; \\ \ \mathtt{pop} : \mathtt{car} \to (\mathtt{car} \times A + \mathit{Unit}) \mid \\ \\ \ \forall s : \mathtt{car} \quad \forall a : A \quad \mathtt{pop}(\mathtt{push}\, s\ a) = \mathtt{inl}\,(s, a) \mid \\ \ \mathtt{pop}(\mathtt{empty}) = \mathtt{inr} \bullet \} \end{array}$$

This definition provides us not only the typing information of the functions `pop` and `push`, but also specify the intended behavior of these functions. An implementation of the stack data type would be an element of the type $\mathit{Stack}(A)$. For example, we can implement stacks as lists:

$$\begin{array}{l} \mathit{stack\_as\_list}(A) \triangleq \\ \{\ \mathtt{car} = A\,\mathrm{List}; \\ \ \ \mathtt{empty} = \mathit{nil}; \\ \ \ \mathtt{push} = \lambda s.\lambda a.a :: s; \\ \ \ \mathtt{pop} = \lambda s.\mathtt{match}\ \ s\ \ \mathtt{with} \\ \qquad\qquad\qquad\quad \mathit{nil} \qquad => \mathtt{inr} \bullet \\ \qquad\qquad\qquad\quad a :: s' \quad => \mathtt{inl}\,(s', a)\} \end{array}$$

The following theorem states that our implementation of stacks indeed satisfy the specification.

Theorem 18 *For any type A*

$$\mathit{stack\_as\_list}(A) \in \mathit{Stack}(A).$$

This theorem was proved in the MetaPRL system.