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

Alexei Kopylov

Department of Computer Science
Cornell University
Ithaca, NY 14853
`kopylov@cs.cornell.edu`

Abstract. Record types are an important tool for programming and dependent record types are proven to be very useful for program specification and verification. Unfortunately all known embedding of the dependent record type in the type theory had some imperfections. In this paper we present a new type constructor, *dependent intersection* — the intersection of *two* types, where the second type may depend on the first type (this is not the intersection of the family of types). This new type constructor allows us to define records that combine the most advantages of previously known approaches, while avoiding most of their disadvantages.

1 Introduction

Over the last decade record types and especially the dependent record types (or more formally dependently typed records) proved to be an extremely useful program specifications and verification tool [7, 3, 11]. There are several approaches of defining record types.

Probably the most straightforward way of adding records to the type theory is to add them as a new primitive type. This approach was taken for example in [8]. Unfortunately, if this approach is taken, the theory needs to be extended with the numerous new rules describing the properties of record types, operations on record types, properties of record objects, operations on record objects, etc. This creates a very complicated theory that is not very intuitive and thus hard reason in and about.

To avoid this problem one can try to define record types using simpler and easier to understand primitives. Jason Hickey [9] based his approach on idea of defining plain records as dependent functions. He introduced a new type of *very depended functions* to represent dependent records. The advantages of this method are the ability to use ordinary λ-expressions to define record objects and a natural subtyping relation between record types. Unfortunately, very dependent functions can not always be easily extended. That is, for example, if

* This work was partially supported by DARPA LPE grant

we have a record type *Monoid* then we cannot use it to define a new record type *Group*. We have to repeat the main part of the definition of *Monoid* in the definition of *Group*. Additionally, very dependent functions is a rather complex concept.

Another way of representing records was used by T. Knoblock and R. Constable in [10] and by R. Pollack in [15]. They defined records in "packaging" approach by using dependent products. This approach provides us with a mechanism for building records from simpler ones as well as the ability to regard record objects as ordinary tuples. This definition does not have structure subtyping, but rather "coercive" subtyping (see [15] for more details).

In this paper we define a new type constructor, *dependent intersection* (Section 2). In Section 3 we show how this new constructor allows us to define record type in the way that combines advantages of the methods outlined above. We build our record type use simple and easy to understand operations. In our approach we can build records incrementally by extending simple ones. Our record types are extensionally equal to Jason Hickey's ones, so we retain the natural subtyping property and the ability to express record objects using λ-terms. In the last section we show the connection between dependent intersections and sets.

2 Dependent Intersection

We introduce a new type constructor *dependent intersection* $x : A \cap B[x]$. This type contains all elements a from A such that a is also in $B[a]$. Do not confuse it with the intersection of the family of types ($\bigcap\limits_{x:A} B[x]$ or $\bigcap x : A.B[x]$)[1].

Example 1. The ordinary binary intersection type is just a special case of a dependent intersection with a constant second argument — $x : A \cap B \ = \ A \cap B$.

Example 2. Let $A = \mathbb{Z}$ and $B[x] = \{y : \mathbb{Z} | y > 2 * x\}$. Then $x : A \cap B[x]$ is a set of all integers, such that $x > 2 * x$.

Semantics We are going to give the formal semantics for this new type constructor based on the predicative PER semantics of Allen [2]. According to Allen, to give the semantics for a new type constructor we need to determine when this constructor is well-formed type and specify the partial equivalence relation that would serve as equality relation for that type.

As usual in Allen's approach, the expression $x : A \cap B[x]$ is a type by definition if and only if A is a type and $B[x]$ is a well-formed type in $x : A$ (that is for any two terms a_1 and a_2 if $a_1 = a_2 \in A$, then the terms $B[a_1]$ and $B[a_2]$ are equal types). Further $a = a' \in x : A \cap B[x]$ iff $a = a' \in A$ and $a = a' \in B[a]$. In particular it means that $a \in (x : A \cap B[x])$ iff $a \in A$ and $a \in B[a]$.

[1] The difference between these two type constructor 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 $\Pi_{x:A} B[x] = x : A \to B[x]$.

The inference rules The inference rules for dependent intersection are presented in Table 1.

$$\frac{\Gamma \vdash A \,\mathrm{Type} \qquad \Gamma; x : A \vdash B[x] \,\mathrm{Type}}{\Gamma \vdash x : A \cap B[x] \,\mathrm{Type}} \qquad (\mathit{TypeFormation})$$

$$\frac{\Gamma \vdash a \in A \qquad \Gamma \vdash a \in B[a] \qquad \Gamma \vdash x : A \cap B[x] \,\mathrm{Type}}{\Gamma \vdash a \in x : A \cap B[x]} \qquad (\mathit{Introduction})$$

$$\frac{\Gamma \vdash a = a' \in A \qquad \Gamma \vdash a = a' \in B[a] \qquad \Gamma \vdash x : A \cap B[x] \,\mathrm{Type}}{\Gamma \vdash a = a' \in x : A \cap B[x]} \qquad (\mathit{Equality})$$

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

Table 1. Inference rules for dependent intersection

3 Records

3.1 Plain Records

The record type is the type of the form

$$\{\mathtt{x}_1 : A_1, \ldots, \mathtt{x}_n : A_n\} \tag{1}$$

where $A_1, \ldots, A_n$ are types and $\mathtt{x}_1, \ldots, \mathtt{x}_n$ are *labels*[2]. Usually labels have string type, but generally speaking labels can be of any fixed type Label with decidable equality. The (1) is the type of all records of the form

$$\{\mathtt{x}_1 = a_1, \ldots, \mathtt{x}_n = a_n\} \tag{2}$$

where a_i has type A_i. The main difference between the record type and the product type is that record type has the *subtyping property:*

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

Example 3. Let $\mathit{Point} = \{\mathtt{x} : \mathit{int}; \mathtt{y} : \mathit{int}\}$ and $\mathit{ColorPoint} = \{\mathtt{x} : \mathit{int}; \mathtt{y} : \mathit{int}; \mathtt{color} : \mathit{Color}\}$. Then the record $\{\mathtt{x} = 0; \mathtt{y} = 0; \mathtt{color} = \mathit{red}\}$ is a $\mathit{ColorPoint}$ and 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 $\mathit{Point} \to T$. Further the result of this function does not depend whether we use $\{\mathtt{x} = 0; \mathtt{y} =$

[2] We will use `truetype font` for labels in contrast to *variables*

$0; \mathtt{color} = red\}$ or $\{\mathtt{x} = 0; \mathtt{y} = 0; \mathtt{color} = green\}$. That is these two records are equal as the elements of the type $Point$, i.e.

$$\{\mathtt{x} = 0; \mathtt{y} = 0; \mathtt{color} = red\} = \{\mathtt{x} = 0; \mathtt{y} = 0; \mathtt{color} = green\} \in \{\mathtt{x} : int; \mathtt{y} : int\} \tag{4}$$

Further, record does not depend on field ordering.

Records as dependent functions Record type can be defined as the dependent function type with the domain Label(cf. [7, 4]):

$$\begin{aligned} \{\mathtt{x}_1 : A_1; \ldots; \mathtt{x}_n : A_n\} = \\ & l : \mathrm{Label} \to \\ & \quad \mathtt{if}\ l = \mathtt{x}_1\ \mathtt{then}\ A_1\ \mathtt{else} \\ & \quad \ldots \\ & \quad \mathtt{if}\ l = \mathtt{x}_n\ \mathtt{then}\ A_n \\ & \quad\ \mathtt{else}\ \mathrm{Top} \end{aligned} \tag{5}$$

Then record objects can be defined as functions:

$$\begin{aligned} \{\mathtt{x}_1 = a_1; \ldots; \mathtt{x}_n = a_n\} = \\ & \lambda l.\mathtt{if}\ l = \mathtt{x}_1\ \mathtt{then}\ a_1\ \mathtt{else} \\ & \quad \ldots \\ & \quad \mathtt{if}\ l = \mathtt{x}_n\ \mathtt{then}\ a_n \end{aligned} \tag{6}$$

And selection is defined as application:

$$r.l = r\ l \tag{7}$$

One can see that this definition meets the natural subtyping properties such as (3) and (4)

Records as Intersections Alternative way to define records is by using intersection. First, let us define single-field record type as the function type:

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

where $\{\mathtt{x}\}$ is the singleton set[3]. This definition is equivalent to the definition (5), although here we uses the functions with domain $\{\mathtt{x}\}$ instead of function with domain Label and range Top for all labels except $\mathtt{x}$.

Now we can define an arbitrary record as the intersection:

$$\{\mathtt{x}_1 : A_1; \ldots; \mathtt{x}_n : A_n\} = \{\mathtt{x}_1 : A_1\} \cap \cdots \cap \{\mathtt{x}_n : A_n\} \tag{9}$$

The record objects and selection operation are defined as before (definitions (6) and (7)).

[3] That is $\{\mathtt{x}\} = \{l : \mathrm{Label} \mid l = \mathtt{x} \in \mathrm{Label}\}$

Example 4. The record $\{\mathtt{x} = 1; \mathtt{y} = 2\}$ by definition (6) 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 is also has type $\{\mathtt{y}\} \to \mathbb{Z} = \{\mathtt{y} : \mathbb{Z}\}$. Hence it has type $\{\mathtt{x} : \mathbb{Z}; \mathtt{y} : \mathbb{Z}\} = \{\mathtt{x} : \mathbb{Z}\} \cap \{\mathtt{y} : \mathbb{Z}\}$.

One can see that when all fields are distinct definitions (5) and (9) 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 internally equal types, i.e. types that have the same elements and the same equality relations.

3.2 Dependent Records

The dependent record type is an expression of the form

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

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

Example 5. $MonoidSig$ is the record type that represents the signature of monoids. It has three fields: a type of carrier $\mathtt{M}$, one picked element $\mathtt{e}$ and a binary operation $\mathtt{product}$:

$$Monoid = \{\mathtt{M} : \mathbb{U}_1; \mathtt{e} : \mathtt{M}; \mathtt{product} : \mathtt{M} \times \mathtt{M} \to \mathtt{M}\}$$

Dependent Records as Very Dependent Functions We can not define dependent record type using "ordinary" dependent function type, because the range of such function type should depend not only on arguments of the function, but also on values of the function. For example, the types of $\mathtt{e}$ and $\mathtt{product}$ fields in the $MonoidSig$ type depends on the value of the field $\mathtt{M}$.

To represent dependent records Jason Hickey [9] introduced the *very dependent function* type constructor:

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

Here A is the domain of 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 example $MonoidSig$ can be represented as a very dependent function type

$$MonoidSig = \{r | l : \mathrm{Label} \to B[r, l]\} \tag{11}$$

where

$$\begin{aligned} B[r, l] = & \mathtt{if}\ l = \mathtt{M}\ \mathtt{then}\ \mathbb{U}_1 \\ & \mathtt{if}\ l = \mathtt{e}\ \mathtt{then}\ r.\mathtt{M} \\ & \mathtt{if}\ l = \mathtt{product}\ \mathtt{then}\ r.\mathtt{M} \to r.\mathtt{M} \to r.\mathtt{M} \\ & \ \mathtt{else}\ \mathrm{Top} \end{aligned}$$

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

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

is not 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 [9] for more details.

Dependent Records as Dependent Intersection By using dependent intersection we can avoid the complex concept of very dependent functions. We can define records simple as dependent intersection of the one-field records:

$$\{\mathtt{x} : A; \mathtt{y} : B[x]; \mathtt{z} : C[x, y]\} = x : \{\mathtt{x} : A\} \cap (y : \{\mathtt{y} : B[x.\mathtt{x}]\} \cap \{\mathtt{z} : C[x.\mathtt{x}, y.\mathtt{y}]\}) \tag{12}$$

Note that here $\mathtt{x}$, $\mathtt{y}$ and $\mathtt{z}$ are labels and x and y are bound variables that are used to refer to the one-field records $\{\mathtt{x} : A\}$ and $\{\mathtt{y} : B[x.\mathtt{x}]\}$ respectively.

For example, $MonoidSig$ can be defined as

$$MonoidSig = M : \{\mathtt{M} : \mathbb{U}_1\} \cap (\{\mathtt{e} : M\} \cap \{\mathtt{product} : M.\mathtt{M} \times M.\mathtt{M} \to M.\mathtt{M}\}) \tag{13}$$

This is a right-associated way of building records: we build them starting from right in the same way as we build dependent products. This way is natural for dependent products, but we usually want something different. Namely, we want to build records by extending the simple ones. We can do that by using left-associating intersections:

$$\{\mathtt{x} : A; \mathtt{y} : B[x]; \mathtt{z} : C[x, y]\} = r_2 : (r_1 : \{\mathtt{x} : A\} \cap \{\mathtt{y} : B[r_1.\mathtt{x}]\}) \cap \{\mathtt{z} : C[r_2.\mathtt{x}, r_2.\mathtt{y}]\} \tag{14}$$

Here r_1 and r_2 are bound variables, r_1 refer to the single-field record $\{\mathtt{x} : A\}$, and r_2 refer to the two-field record $\{\mathtt{x} : A; \mathtt{y} : B[r_1.\mathtt{x}]\}$. Note that the scopes of these variables do not intersect (r_1 is bound in B and r_2 is bound in C). Therefore it is convenient to use the same name $self$ for them.

Robert Pollack [15] considered two types of depended records: left associating records and right associating records. But here left and right associating are just two different ways of building records. Record types defined by (12) and by (14) turn out to be equal. And if all fields have distinct names these types are equal to the Hickey's records defined as very dependent functions.

For example the following definitions of $MonoidSig$ is equivalent to the definitions above ((11) and (13)).

$$\begin{aligned} MonoidSig = self : (self : \{\mathtt{M} : \mathbb{U}_1\} \cap \{\mathtt{e} : self.\mathtt{M}\}) \cap \\ \{\mathtt{product} : self.\mathtt{M} \times self.\mathtt{M} \to self.\mathtt{M}\} \end{aligned}$$

We can also combine left and right associating. For example, we can extend $MonoidSig$ to $Monoid$, $GroupSig$ and $Group$ in the following way:

$$GroupSig = self : MonoidSig \cap \{\texttt{inv} : self.\texttt{M} \to self.\texttt{M}\}$$

$$\begin{aligned}
Monoid &= self : MonoidSig \cap \\
&\quad (\{\texttt{ax_neutral} : \forall x : self.\texttt{M} \;\; x \cdot self.\texttt{e} = x\} \cap \\
&\quad \{\texttt{ax_associativity} : \forall x, y, z : self.\texttt{M} \;\; (x \cdot y) \cdot z = x \cdot (y \cdot z)\}) \\
Group &= self : (GroupSig \cap Monoid) \cap \\
&\quad \{\texttt{ax_inv} : \forall x, y : self.\texttt{M} \;\; (x \cdot y^{-1}) = self.\texttt{e}\}
\end{aligned}$$

Here $x \cdot y$ is an abbreviation of $self.\texttt{product}\, x\, y$ and x^{-1} refers to $self.\texttt{inv}\, x$.

Note that we use both $Monoid$ and $GroupSig$ in the definition of $Group$.

4 Dependent Intersection and Sets

Dependent intersection is in some sense similar to sets. Namely, one can image $x : A \cap B[x]$ as the set $\{x : A | x \in B[x]\}$. But this set is almost never going to be a type in Martin-Löf's type theory. Indeed this set is a type only when for any x from A the expression $x \in B[x]$ is a type. But this is true only in a trivial case when $x \in B[x]$ is true for all $x \in A$.

That means that we can not define dependent intersections using sets. But there is a connection between sets and dependent intersection. Let $\mathrm{squash}(P) = \{\mathrm{Top} | P\}$. Then

$$\{x : A | P[x]\} = x : A \cap \mathrm{squash}(P[x])$$

So if we treat squash as the primitive connective, then we can define sets using squash and dependent intersection[4].

5 Acknowledgements

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

References

1. Stuart F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language.* PhD thesis, Cornell University, 1987.
2. Stuart F. Allen. A Non-type-theoretic Definition of Martin-Löf's Types. In *Proceedings of the Second Symposium on Logic in Computer Science*, pages 215–224. IEEE, June 1987.

[4] If we add some principles (such as Markov principle) in the type theory, then squash(P) can be defined as double negation of P.

3. 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.
4. Robert L. Constable. *Types in Logic, Mathematics and Programming*, chapter 10. Elsevier Science, 1997.
5. Robert L. Constable et al. *Implementing Mathematics with the NuPRL Development System*. Prentice-Hall, 1986.
6. Robert L. Constable and Jason Hickey. Nuprl's class theory and its applications. *Foundations of Secure Computation NATO Science*, Series F:91–116, 2000.
7. Karl Crary. Technical Report CMU-CS-94-100, Carnegie Mellon University, Pittsburgh, PA, 1999.
8. Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando FL*, pages 131–142. ACM, January 1991. Extended version available as Carnegie Mellon Technical Report CMU-CS-90-157.
9. Jason J. Hickey. Objects and theories as very dependent types. In *Proceedings of FOOL 3*, July 1996.
10. 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.
11. Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason J. Hickey, Mark Hayden, Kenneth Birman, and Robert Constable. Building reliable, high-performance communication systems from components. In *17^{th} ACM Symposium on Operating Systems Principles*, December 1999.
12. 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.
13. Benjamin C. Pierce. Preliminary investigation of a calculus with intersection and union types. Unpublished manuscript, June 1990.
14. Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, February 1991.
15. Robert Pollack. Dependently typed records for representing mathematical structure. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 461–478. Springer-Verlag, 2000.