A Type Theory with Partial Equivalence Relations as Types

Abhishek Anand  Mark Bickford  Robert L. Constable
Vincent Rahli

May 13, 2014
This work started with a careful reading of:

Stuart Allen’s PhD thesis [All87]:

**A Non-Type-Theoretic Semantics for Type-Theoretic Language**

It describes a semantics for Nuprl where types are defined as Partial Equivalence Relations on terms (the PER semantics).
Stuart Allen’s Thesis

Among others, Nuprl has the following types:

**Equality:** $a = b \in T$

**Dependent function:** $a : A \rightarrow B[a]$

**Dependent product:** $a : A \times B[a]$

**Intersection:** $\cap a : A. B[a]$

**Partial:** $\overline{A}$

**Universe:** $\mathbb{U}$_i

**Subset:** $\{ a : A \mid B[a] \}$

**Quotient:** $T // E$

where $E$ has to be an equivalence relation w.r.t. $T$. 
forming an \( a \in A \) such that \( B[a/x] \) is inhabited; two equal canonical members are formed by forming \( a, a' \in \{ x \in A \mid B \} \) such that \( E[a, a'/u, v] \) is inhabited. The set type and quotient type constructors could have been unified in a single constructor \( x, y \in A/E_{xy} \) which is like quotient except that, rather than requiring (the inhabitation of) \( E_{xy} \) to be an equivalence relation, we require only that it be transitive and symmetric over \( A \), i.e., its restriction to \( A \) should be a partial equivalence relation. The equal members are the members of \( A \) that make \( E_{xy} \) inhabited. Thus, a type \( x, y \in A/E_{xy} \) is extensionally equal to \( x, y \in A/E_{xy} \), and a type \( \{ x \in A \mid B_x \} \) is extensionally equal to \( x, y \in A/(B_x \times I(A, x, y)) \).

We come now to Nuprl’s treatment of assumptions. Nuprl uses one form of judgement:

\[ \exists_1 \in A \ldots \exists_n \in A_n \Rightarrow t \in T. \]

Let us start by considering Nuprl judgements with one assumption. The meaning of \( \exists x \in A \Rightarrow t \in T \) is that, for any \( a \) and \( a' \), if \( a = a' \in A \) then \( T[a/x] = T[a'/x] \) and \( t[a/x] = t[a'/x] \in T[a/x] \). Notice that, rather than implying or presupposing that \( A \) is a type, the typehood of \( A \) is part of the assumption (since the typehood of \( A \) is implied by \( a = a' \in A \) ). Thus, if \( A \) cannot be defined as a type, because it has no value, say, then we may infer for any \( x, T, \) and \( t \) that \( x \in A \Rightarrow t \in T \). In contrast, we cannot infer \( t \in T (x \in A) \) unless we also know that \( A \) is a type. Since we are discussing two forms of assumption, it will be convenient to introduce a distinguishing nomenclature; there will be no need to make the general application of the terminology precise. We shall say an assumption \( x \in A \) is positive within the judgements that, by virtue of that assumption, imply the typehood of \( A \), and we shall say the assumption is negative within the judgements in which the typehood of \( A \) is a part of what \( x \) is being assumed. The assumption \( x \in A \) is positive within \( t \in T (x \in A) \) and negative within \( x \in A \Rightarrow t \in T \). The use of negative assumptions allows one to express the assumption that \( a \) is a member of \( A \) as a negative assumption \( x \in I(A, a, a) \). A positive assumption of this form would be vacuous since for \( I(A, a, a) \) to be a type \( A \) must be a type with member \( a \).

Now we shall consider judgements that use two negative assumptions. The meaning intended for judgements using more assumptions should be clear in light of the explanation for two assumptions. A coarse reading, one

\[ 20 \] The notation used in [Constable et al. 86] is

\[ \exists_1 : A_1 \ldots \exists_n : A_n \Rightarrow T \text{ ext } t. \]

The part “ext” is not displayed by the Nuprl system when it occurs in proofs, but rather, it is extracted from a completed proof. Most proofs are constructed without the user knowing precisely what term is to be extracted.
What does it say?

It suggests that the *quotient* and *subset* types could be replaced by a quotient-like type that only requires a partial equivalence relation.
Our Proposal

Here is our proposal—redefining Nuprl’s type theory around an **extensional “Partial Equivalence Relation” type constructor** that turns PERs into types.

The domain: the closed terms of Nuprl’s computation system.

**Base** is the type that contains all closed terms and whose equality \( \sim \) is Howe’s computational equivalence relation [How89].
Our Proposal

Now, the \textit{per} type constructor:

\begin{itemize}
  \item per(\(R\)) is a type if \(R\) is a \textbf{PER on} Base.
  \item \(a = b \in \text{per}(\(R\))\) if \(R\ a b\).
  \item per(\(R_1\)) = per(\(R_2\)) \(\in \mathbb{U}_i\) if \(R_1\) and \(R_2\) are equivalent relations.
\end{itemize}

We’ll need universes as well.

\textbf{Our type theory now has:} Base, \(\mathbb{U}_i\), per.
Our Proposal

Per types are now part of our implementation of Nuprl in Coq [AR14]. We verified:

\[ H \vdash \text{per}(R) = \text{per}(R') \in \text{Type} \]
\[ \text{BY \ [pertypeEquality]} \]
\[ H, x : \text{Base}, y : \text{Base} \vdash R \times y \in \text{Type} \]
\[ H, x : \text{Base}, y : \text{Base} \vdash R' \times y \in \text{Type} \]
\[ H, x : \text{Base}, y : \text{Base}, z : R \times y \vdash R' \times y \]
\[ H, x : \text{Base}, y : \text{Base}, z : R' \times y \vdash R \times y \]
\[ H, x : \text{Base}, y : \text{Base}, z : R \times y \vdash R' \times y \]
\[ H, x : \text{Base}, y : \text{Base}, z : \text{Base}, u : R \times y, v : R \times z \vdash R \times z \]

\[ H, x : t_1 = t_2 \in \text{per}(R) \vdash C [\text{ext e}] \]
\[ \text{BY \ [pertypeElimination]} \]
\[ H, x : t_1 = t_2 \in \text{per}(R), [y : R \ t_1 \ t_2] \vdash C [\text{ext e}] \]

\[ H \vdash t_1 = t_2 \in \text{per}(R) \]
\[ \text{BY \ [pertypeMemberEquality]} \]
\[ H \vdash \text{per}(R) \in \text{Type} \]
\[ H \vdash R \ t_1 \ t_2 \]
\[ H \vdash t_1 \in \text{Base} \]
\[ H \vdash t_2 \in \text{Base} \]
Examples

Let us start with simple examples:

\[ \text{Void} = \text{per}(\lambda _, _\cdot 1 \preceq 0) \]

\[ \text{Top} = \text{per}(\lambda _, _\cdot 0 \preceq 0) \]

These use \( \preceq \), Howe’s computational approximation relation [How89].

Our type theory now has: Base, \( \mathbb{U}_i \), per, \( \preceq \).
Examples

Integers:

\[ \mathbb{Z} = \text{per}(\lambda a. \lambda b. a \sim b \sqcap \uparrow(\text{isint}(a, \text{tt}, \text{ff}))) \]

where

\[ \text{A} \sqcap \text{B} = \sqcap x : \text{Base}. \sqcap y : \text{halts}(x). \text{isaxiom}(x, \text{A}, \text{B}) \]

\[ \uparrow(a) = \text{tt} \preceq a \]

\[ \text{halts}(t) = \text{Ax} \preceq (\text{let } x := t \text{ in Ax}) \]

Our type theory now has: Base, \( \mathbb{U} \), per, \( \preceq \), \( \sim \), \( \sqcap \).
Examples

Quotient types:

\[ T /\!\!\!\!/ E = \text{per}(\lambda x, y. (x \in T) \sqcap (y \in T) \sqcap (E \times y)) \]

This is the definition we are using in Nuprl now—no longer a primitive.

The partial type constructor is a quotient type—no longer a primitive.

Our type theory now has: Base, \( \mathbb{U} \), per, \( \preceq \), ~, \( \cap \), _ = _, _ \in _. 
What about the subset type?

\[
\{ a : A \mid B[a] \} = \text{per}(\lambda x, y. (x = y \in A) \sqcap B[x])
\]
What about the subset type?

\[ \{ a : A \mid B[a] \} = \text{per}(\lambda x, y. (x = y \in A) \sqcap B[x]) \]

This does not work!

We do not get that \( B \) is functional over \( A \).
Examples

one solution—annotate families with levels:

\[ \{ a : A \mid B[a] \} ; = \text{per}(\lambda x, y. (x = y \in A) \cap B[x] \cap Fam(A, B, i)) \]

where

\[ Fam(A, B, i) = \cap a, b : A. (B[a] = B[b] \in \mathbb{U}_i) \]

One drawback: the annotations.
another solution—introduce a type of type equalities ($T = U$):

$$\{a : A \mid B[a]\} = \text{per}(\lambda x, y.(x = y \in A) \sqcap B[x] \sqcap \text{Fam}(A, B))$$

where

$$\text{Fam}(A, B) = \sqcap a, b:A.(B[a] = B[b])$$

This requires a more intensional version of our per type.
Examples

Using this method, we can also define the other type families such as: \textit{dependent functions}, dependent products, \ldots

Both \texttt{per} and its intensional version are part of our implementation of Nuprl in Coq [AR14].

We proved, e.g., that the elimination rule for the \texttt{per} version of our function type is valid.
Inductive types

We saw how to build inductive types in yesterday's talk.

- Algebraic datatypes: \( \{ t : coDT \mid \text{halts}(\text{size}(t)) \} \).

- Inductive types using Bar Induction.
Conclusion

- **Conciseness**
  - A small core of primitive types.
  - Simple rules.

- **Flexibility**
  - Lets user define even more types.
  - No need to modify/update the meta-theory.

- **Practicality?**
  - We’re already using it.
  - We’re still experimenting with the intensional per type.
Stuart F. Allen.
A Non-Type-Theoretic Semantics for Type-Theoretic Language.

Abhishek Anand and Vincent Rahli.
Towards a formally verified proof assistant.

Douglas J. Howe.
Equality in lazy computation systems.