A Type Theory with Partial Equivalence Relations as Types

Abhishek Anand   Mark Bickford   Robert L. Constable
Vincent Rahli

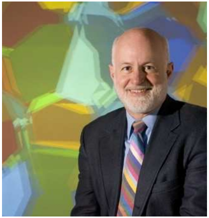
May 13, 2014

PRL Group

Abhishek Anand

Mark Bickford

Robert L. Constable



Richard Eaton

Vincent Rahli

Stuart Allen's Thesis

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.

Stuart Allen's Thesis

In his thesis, the following page was misplaced:

1.2 The Type Theory of Nuprl 13

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 $E[a, a'/x, x']$ is inhabited. The set type and quotient type constructors could have been unified in a single constructor $x, y \in A//E_{x,y}$ which is like quotient except that, rather than requiring (the inhabitation of) $E_{x,y}$ 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 $E_{x,y}$ inhabited. Thus, a type $x, y \in A//E_{x,y}$ is extensionally equal to $x, y \in A/E_{x,y}$, and a type $\{ x \in A \mid B_x \}$ is extensionally equal to $x, y \in A//(B_x \times \mathrm{I}(A, x, y))$.

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

$$x_1 \in A_1 \ldots x_n \in A_n \gg t \in T.^{20}$$

Let us start by considering Nuprl judgements with one assumption. The meaning of $x \in A \gg t \in T$ 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 $x \in A \gg t \in T$. 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 is being assumed. The assumption $x \in A$ is positive within $t \in T$ $(x \in A)$ and negative within $x \in A \gg t \in T$. The use of negative assumptions allows one to express the assumption that a is a member of A as a negative assumption $x \in \mathrm{I}(A, a, a)$. A positive assumption of this form would be vacuous since for $\mathrm{I}(A, a, a)$ 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

$$x_1 : A_1 \ldots x_n : A_n \gg T \text{ ext } t.$$

The part "ext t" 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.

Stuart Allen's Thesis

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 **per** type constructor:

- $\mathtt{per}(R)$ is a type if R **is a PER on** $\mathtt{Base}$.
- $a = b \in \mathtt{per}(R)$ if $R\ a\ b$.
- $\mathtt{per}(R_1) = \mathtt{per}(R_2) \in \mathbb{U}_i$ if R_1 and R_2 are equivalent relations.

We'll need universes as well.

Our type theory now has: $\mathtt{Base}$, $\mathbb{U}_i$, $\mathtt{per}$.

Our Proposal

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

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

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

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

Examples

Let us start with simple examples:

$$\texttt{Void} = \texttt{per}(\lambda\_, \_.1 \preceq 0)$$

$$\texttt{Unit} = \texttt{per}(\lambda\_, \_.0 \preceq 0)$$

These use $\preceq$, Howe's computational approximation relation [How89].

Our type theory now has: $\texttt{Base}$, $\mathbb{U}_i$, $\texttt{per}$, $\preceq$.

Examples

Integers:

$$\mathbb{Z} = \mathtt{per}(\lambda a.\lambda b.a \sim b \sqcap \Uparrow(\mathtt{isint}(a, \mathtt{tt}, \mathtt{ff})))$$

where

$$A \sqcap B = \cap x{:}\mathtt{Base}. \cap y{:}\mathtt{halts}(x).\mathtt{isaxiom}(x, A, B)$$

$$\Uparrow(a) = \mathtt{tt} \preceq a$$

$$\mathtt{halts}(t) = \mathtt{Ax} \preceq (\mathtt{let}\ x := t\ \mathtt{in}\ \mathtt{Ax})$$

Our type theory now has: $\mathtt{Base}$, $\mathbb{U}_i$, $\mathtt{per}$, $\preceq$, $\sim$, $\cap$.

Examples

Quotient types:

$$T//E = \mathtt{per}(\lambda x, y.(x \in T) \sqcap (y \in T) \sqcap (E\ x\ y))$$

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: $\mathtt{Base}$, $\mathbb{U}_i$, $\mathtt{per}$, $\preceq$, $\sim$, $\cap$, $\_ = \_ \in \_$.

Examples

What about the subset type?

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

Examples

What about the subset type?

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

This does not work!

We do not get that B is functional over A.

Examples

one solution—annotate families with levels:

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

where

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

One drawback: the annotations.

Examples

another solution—introduce a type of type equalities ($T = U$):

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

where

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

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

Examples

Using this method, we can also define the other type families such as: **dependent functions**, dependent products, ...

Both `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 `per` version of our function type is valid.

Inductive types

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

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

References I

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

Abhishek Anand and Vincent Rahli.
Towards a formally verified proof assistant.
Accepted to ITP 2014, 2014.

Douglas J. Howe.
Equality in lazy computation systems.
In *Proceedings of Fourth IEEE Symposium on Logic in Computer Science*, pages 198–203. IEEE Computer Society, 1989.