A Type Theory with Partial Equivalence Relations as Types

Abhishek Anand, Mark Bickford, Robert L. Constable, and Vincent Rahli

Cornell University

Abstract

A small core type language with intersection types in which a partial equivalence relation on closed terms is a type is enough to build the non-inductive types of Nuprl, including the types of dependent functions and partial functions. Using induction on natural numbers and intersection types, we build coinductive types; and using partial functions and coinductive types we build algebraic datatypes.

Introduction. Nuprl [6, 2] is a functional programming language based on a constructive dependent type theory with partial types called CTT. As in similar systems such as Coq [4] and Agda [5], it has dependent functions, inductive types, and a cumulative hierarchy of universes. In addition, CTT has dependent products, disjoint union, integer,[1], equality, set (or refinement) and quotient types [6]; intersection and union types [10]; image types [11]; computational approximation and equivalence types [12]; and is one of the only type theories with partial types [7, 8].

Allen gave a semantics of CTT where a type is a Partial Equivalence Relation (PER) on closed terms [1], which is connected to Russell's original definition of a type as "the range of significance of a propositional function." By allowing the theory to directly represent PERs as types, we can reformulate CTT using a smaller core of primitive type constructors. For example, the dependent function type can now be defined. Allen [1, pp.15] suggested such a type that represents PERs by combining the set and quotient types.

The `per` type constructor can turn PERs into types. Therefore, we need some primitives to express such PERs: `Base` is the type of closed terms (PERs are relations on closed terms) whose equality $\sim$ is Howe's computational equivalence [9]; equality (or identity) types to refer to already defined PERs[2]; our main logical operator is the intersection type constructor which is a uniform universal quantifier; the computational approximation type constructor $\preceq$ allows us to build PERs by imposing restrictions on their domains in terms of how terms compute.

When the partial, union and image types were added to Nuprl in the past we had to update the metatheory accordingly. Using the `per` constructor we can now add new types to Nuprl without changing the metatheory. We are already using this type in Nuprl and have defined several formerly primitive types using it, such as the quotient and partial types.

Nuprl's syntax. Nuprl is defined on top of an applied lazy untyped λ-calculus. We define the subset of this language that is of interest to us in this paper as follows:

$$
\begin{array}{c}
A, B, R ::= \ t_1 \preceq t_2 \mid \ \mathtt{Base} \mid \ \mathbb{U}_i \mid \ \mathtt{per}(R) \mid \ \cap\, x{:}A.B[x] \mid \ t_1 = t_2 \in A \\
v ::= \ A \mid \ \underline{i} \mid \ \lambda x.t \mid \ \langle t_1, t_2 \rangle \mid \ \mathtt{Ax} \mid \ \mathtt{inl}(t) \mid \ \mathtt{inr}(t) \\
t ::= x \mid \ v \mid \ t_1\ t_2 \mid \ \mathtt{fix}(t) \mid \ \mathtt{let}\ x, y = t_1\ \mathtt{in}\ t_2 \mid \ \mathtt{let}\ x := t_1\ \mathtt{in}\ t_2 \\
\mid \mathtt{if}\ t_1 {<} t_2\ \mathtt{then}\ t_3\ \mathtt{else}\ t_4 \mid \ \mathtt{isint}(t_1, t_2, t_3) \mid \ \mathtt{isaxiom}(t_1, t_2, t_3)
\end{array}
$$

where A, B, and R stand for types, $\underline{i}$ for an integer, v for a value, x for a variable, and t for a term. `Ax` is the unique canonical inhabitant of true propositions that do not have any nontrivial computational meaning in CTT, such as $0 = 0 \in \mathbb{N}$. The canonical form tests such as

[1] For efficiency issues, the integer type is a primitive type in Nuprl.

[2] We extended the definition of equality types so that the equality in T is not only a relation on T but also a relation on `Base` [3, Sec. 4.2.1].

$\texttt{isaxiom}$ allow us to distinguish between the different canonical forms [12]. A term of the form $\texttt{let}\ x := t_1\ \texttt{in}\ t_2$ eagerly evaluates t_1 before evaluating t_2.

The Booleans are: $\texttt{tt} = \texttt{inl}(\texttt{Ax})$ and $\texttt{ff} = \texttt{inr}(\texttt{Ax})$. The following operation lifts Booleans to propositions: $\Uparrow(a) = \texttt{tt} \preceq a$, which implies that a is computationally equivalent to $\texttt{tt}$. The following operator asserts that its parameter computes to a value: $\texttt{halts}(t) = \texttt{Ax} \preceq (\texttt{let}\ x := t\ \texttt{in}\ \texttt{Ax})$. We define the following uniform implication: $A \Rrightarrow B = \cap x{:}A.B$, where x does not occur free in B; uniform and: $A \sqcap B = \cap x{:}\texttt{Base}. \cap y{:}\texttt{halts}(x).\texttt{isaxiom}(x, A, B)$; uniform iff: $A \Lleftarrow\!\!\Rrightarrow B = (A \Rrightarrow B \sqcap B \Rrightarrow A)$; computational equivalence: $t_1 \sim t_2 = t_1 \preceq t_2 \sqcap t_2 \preceq t_1$.

Meaning of `per` types. A term of the form $\texttt{per}(R)$ is a type if for all closed terms t_1 and t_2, $R\ t_1\ t_2$ is a type, and R is a PER on closed terms. Two $\texttt{per}$ types $\texttt{per}(R_1)$ and $\texttt{per}(R_2)$ are equal if for all closed terms t_1 and t_2, $R_1\ t_1\ t_2$ is inhabited iff $R\ t_1\ t_2$ is inhabited. Two terms t_1 and t_2 are equal in $\texttt{per}(R)$ if $R\ t_1\ t_2$ is inhabited. We have formally proved in our Coq metatheory that the derivation rules that implement these conditions are valid [3, Sec. 5.2.4].

Type definitions. We now show how one defines Nuprl's partial and function types using the core type system described above. We first start with the simple $\texttt{Void}$, $\texttt{Unit}$ and $\mathbb{Z}$ types.

$$\texttt{Void} = \texttt{per}(\lambda a.\lambda b.\texttt{tt} \preceq \texttt{ff}) \qquad \texttt{Unit} = \texttt{per}(\lambda a.\lambda b.\texttt{tt} \preceq \texttt{tt})$$
$$\mathbb{Z} = \texttt{per}(\lambda a.\lambda b.a \sim b \sqcap \Uparrow(\texttt{isint}(a, \texttt{tt}, \texttt{ff})))$$
$$a{:}A \to B[a] = \texttt{per}(\lambda f.\lambda g. \cap a, b{:}\texttt{Base}.a = b \in A \Rrightarrow f\ a = g\ b \in B[a])$$
$$\overline{A} = \texttt{per}(\lambda x, y.(\texttt{halts}(x) \Lleftarrow\!\!\Rrightarrow \texttt{halts}(y)) \sqcap (\texttt{halts}(x) \Rrightarrow x = y \in A) \sqcap \cap a{:}\texttt{Base}.a \in A \Rrightarrow \texttt{halts}(a))$$

Using these definitions, several of our inference rules can be proved as lemmas.

Algebraic datatypes. Let $\mathbb{N} = \texttt{per}(\lambda a.\lambda b.a = b \in \mathbb{Z} \sqcap \Uparrow(\texttt{if}\ -1 < a\ \texttt{then}\ \texttt{tt}\ \texttt{else}\ \texttt{ff}))$. We assume the existence of an induction principle on $\mathbb{N}$. Using induction on $\mathbb{N}$ and intersection types, we build coinductive types: $\texttt{corec}(G) = \cap n{:}\mathbb{N}.\texttt{fix}(\lambda P.\lambda n.\texttt{if}\ n{=}0\ \texttt{then}\ \texttt{Top}\ \texttt{else}\ G\ (P\ (n{-}1)))\ n$; and using partial functions and coinductive types we build algebraic datatypes. (In order to build inductive types we can add W types to our core system. However, in a companion paper we discuss how to build inductive types using Bar Induction instead.) Our method consists in selecting the largest collection of terms on which the subterm relation is well-founded. We then derive induction principles using this selection procedure. Given a coalgebraic datatype T, we define a size function s on T. Using fixpoint induction [8] we can prove that for all $t \in T$, $s(t) \in \overline{\mathbb{Z}}$. We can then prove that $(\exists n : \mathbb{N}.\ s(t) = n \in \overline{\mathbb{Z}}) \in \mathbb{P}$. We define our algebraic datatype as $\{t : T \mid \exists n : \mathbb{N}.\ s(t) = n \in \overline{\mathbb{Z}}\}$. To prove inductive properties of algebraic datatypes, we can then go by induction on n.

References

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

[2] Stuart F. Allen, Mark Bickford, Robert L. Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and Evan Moran. Innovations in computational type theory using Nuprl. *J. Applied Logic*, 4(4):428–469, 2006. `http://www.nuprl.org/`.

[3] Abhishek Anand and Vincent Rahli. Towards a formally verified proof assistant. Technical report, Cornell University, 2014. `http://www.nuprl.org/html/verification/`.

[4] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development.* SpringerVerlag, 2004. `http://coq.inria.fr/`.

[5] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda - a functional language with dependent types. In *Theorem Proving in Higher Order Logics, 22nd Int'l Conf.*, volume 5674 of *LNCS*, pages 73–78. Springer, 2009. `http://wiki.portal.chalmers.se/agda/pmwiki.php`.

[6] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[7] Robert L. Constable and Scott F. Smith. Partial objects in constructive type theory. In *LICS*, pages 183–193. IEEE Computer Society, 1987.

[8] Karl Crary. *Type-Theoretic Methodology for Practical Programming Languages.* PhD thesis, Cornell University, Ithaca, NY, August 1998.

[9] 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.

[10] Alexei Kopylov. *Type Theoretical Foundations for Data Structures, Classes, and Objects.* PhD thesis, Cornell University, Ithaca, NY, 2004.

[11] Aleksey Nogin and Alexei Kopylov. Formalizing type operations using the "image" type constructor. *Electr. Notes Theor. Comput. Sci.*, 165:121–132, 2006.

[12] Vincent Rahli, Mark Bickford, and Abhishek Anand. Formal program optimization in Nuprl using computational equivalence and partial types. In *ITP 2013*, volume 7998 of *LNCS*, pages 261–278. Springer, 2013.