RedPRL is an experimental proof assistant based on Cartesian cubical computational type theory, a new type theory for higher-dimensional constructions inspired by homotopy type theory. In the style of Nuprl, RedPRL users employ tactics to establish behavioral properties of cubical functional programs embodying the constructive content of proofs. Notably, RedPRL implements a two-level type theory, allowing an extensional, proof-irrelevant notion of exact equality to coexist with a higher-dimensional proof-relevant notion of paths.

1 Introduction

Homotopy type theory [27] and Univalent Foundations [29] extend traditional type theory with a number of axioms inspired by homotopy-theoretic models [19], namely Voevodsky’s univalence axiom [28] and higher inductive types [21]. In recent years, these systems have been deployed as algebraic frameworks for formalizing results in synthetic homotopy theory [8, 12, 17], sometimes even leading to the discovery of novel generalizations of classical theorems [1].

The constructive character of type theory, embodied in the existence of canonical forms, is disrupted by axiomatic extensions that are not accounted for in the traditional computational semantics. Far from being merely a philosophical concern, the negation of type theory’s native algorithmic content impacts the practice of formalization. Brunerie proved that the 4th homotopy group of the 3-sphere is isomorphic to \( \mathbb{Z}/n\mathbb{Z} \) for some closed \( n \), but establishing \( n = 2 \) required years of additional investigation [11]. In ordinary type theory, such a closed \( n \) expresses an algorithm that calculates a numeral; in the absence of computational semantics for homotopy type theory, there is no reason to expect Brunerie’s proof to specify such an algorithm.

Multiple researchers have recently established the constructivity of univalence and higher inductive types by extending the syntax and semantics of type theory with cubical machinery. One solution is embodied in the (De Morgan) cubical type theory of Cohen et al. [14], implemented in the cubicaltt type checker [15]. RedPRL is an interactive proof assistant based on another approach, Cartesian cubical computational type theory [3, 4, 13].

In Section 2 we discuss the methodology of computational type theory, as pioneered in the Nuprl system [16], and in Section 5 we describe its proof-theoretic realization in RedPRL. In Section 4 we briefly discuss the cubical machinery present in Cartesian cubical computational type theory, and its RedPRL implementation. Finally, in Section 5 we discuss related and future work.
Expressions: \( M, N, O \equiv x \mid (x : M) \to N \mid \lambda x . M \mid MN \mid (x : M) \times N \mid \langle M, N \rangle \mid \text{fst}(M) \mid \text{snd}(M) \mid \text{bool} \mid \text{true} \mid \text{false} \mid \text{if}(M; N; O) \)

\[
\begin{array}{c|c|c|c|c}
(x : M) \to N & \lambda x . M & MN & (\lambda x . M) N & (x : M) \times N \\
\text{val} & \text{val} & \text{val} & (\lambda x . M) N \mapsto M[N/x] & \text{val} \\
\hline
\langle M, N \rangle & \text{fst}(M) & \text{snd}(M) & \text{fst}(\langle M, N \rangle) \mapsto M & \text{snd}(M) \mapsto \text{snd}(M') \\
\text{val} & \text{val} & \text{val} & \text{val} & \text{val} \\
\hline
\text{snd}(\langle M, N \rangle) \mapsto N & \text{bool} & \text{true} & \text{false} & \text{if}(M; N; O) \mapsto \text{if}(M'; N; O) \\
\hline
\text{if}(\text{true}; N; O) \mapsto N & \text{if}(\text{false}; N; O) \mapsto O
\end{array}
\]

Figure 1: A programming language with dependent functions, dependent pairs, and booleans.

## 2 Computational Type Theory

RedPRL and Nuprl are based on a computation-first methodology in which the judgments of type theory range over programs from a programming language whose syntax and dynamics have already been defined. This is in contrast with other approaches in which proof terms are inductively defined and equipped after the fact with a reduction semantics or realizability model. One advantage of our methodology is the ability to incorporate features found in computer science more easily, such as exceptions and non-termination, because the theories arise directly from programming languages.

Consider the programming language whose grammar and small-step dynamics are specified in Figure 1; the dynamics consists of a stepping relation \( M \mapsto M' \) and a value predicate \( M \mapsto \text{val} \). In computational type theory, types are interpreted not as structured sets, but as behaviors specifying classes of programs. To define such a type theory over this language, we must specify which programs name types (e.g., \( \text{bool} \)), and which programs each type classifies (e.g., those that evaluate to \( \text{true} \) or \( \text{false} \)). More precisely, we must consider what types are and when they are equal—in which case they must classify the same programs—and for each type, what it means for its elements to evince equal behaviors \([22]\).

Programs \( A \) and \( B \) are equal types, written \( A \equiv B \) type, when \( A \Downarrow A_0 \) (i.e., \( A \mapsto^* A_0 \) and \( A_0 \mapsto \text{val} \)), \( B \Downarrow B_0 \), and \( A_0 \) and \( B_0 \) are equal canonical types. A canonical type \( A_0 \), in turn, is a value associated to a (symmetric and transitive) relation on values, specifying the canonical elements of \( A_0 \) and when two such elements are equal. Finally, \( M \) and \( N \) are equal elements of \( A \equiv A \) type, written \( M \equiv N \in A \), when \( M \Downarrow M_0 \), \( N \Downarrow N_0 \), and \( M_0 \) and \( N_0 \) are classified as equal elements by \( A_0 \) (where \( A \Downarrow A_0 \)). For notational convenience, we write \( A \) type when \( A \equiv A \) type, and \( M \in A \) when \( M \equiv M \in A \).

Concretely, the booleans are defined by the following clauses:

\[
\text{bool} \equiv \text{bool} \quad \text{true} \equiv \text{true} \in \text{bool} \quad \text{false} \equiv \text{false} \in \text{bool}
\]

That is, \( \text{bool} \) is a canonical type with canonical (unequal) elements \( \text{true} \) and \( \text{false} \). The canonicity property follows directly from the above definitions.
Theorem 1 (Canonicity). For any \( M \in \text{bool} \), \( M \Downarrow \text{true} \) or \( M \Downarrow \text{false} \).

The judgments for programs with free variables are defined in terms of all closing substitutions. In the case with one variable, we say \( x : A \Rightarrow B \equiv C \) type when for all \( M \) and \( N \), if \( M \Downarrow N \in A \) then \( B[M/x] \Downarrow C[N/x] \) type. That is, the type families \( B \) and \( C \) are equal when they send equal elements of \( A \) to equal types. Similarly, we say \( x : A \Rightarrow M \Downarrow N \in B \) when for all \( O \) and \( P \), if \( O \Downarrow P \in A \) then \( M[O/x] \Downarrow N[P/x] \in B[O/x] \). Dependent function and dependent pair types are then defined by the following clauses:

\[
(x : A) \rightarrow B \equiv (x : C) \rightarrow D \text{ type } \iff A \equiv C \text{ type and } x : A \Rightarrow B \equiv D \text{ type}
\]

\[
\lambda x.M \equiv \lambda x.N \in (x : A) \rightarrow B \iff x : A \Rightarrow M \Downarrow N \in B
\]

\[
(x : A) \times B \equiv (x : C) \times D \text{ type } \iff A \equiv C \text{ type and } x : A \Rightarrow B \equiv D \text{ type}
\]

\[
\langle M,N \rangle \equiv \langle O,P \rangle \in (x : A) \times B \iff M \Downarrow O \in A \text{ and } N[M/x] \Downarrow P[O/x] \in B[M/x]
\]

In general, new type constructors are implemented by extending the syntax and dynamics of the programming language, and adding clauses to the definitions of canonical types and elements. For the full programming language currently in RedPRL and the technical details of the defining clauses, see Angiuli et al. [4] and Cavallo and Harper [13].

Open judgments in computational type theory are distinct from those in many type theories: variables range over closed programs, instead of being indeterminate objects (sometimes called “generic values”). One consequence is that computational type theory often validates relatively strong extensionality principles, including the full universal property of the \( \text{bool} \) type:

\[
x : \text{bool} \Rightarrow \mathcal{E}[x] \equiv \mathcal{E}[(x; \text{true}; \text{false})] \in \mathcal{F}[x]
\]

where \( \mathcal{E}[-] \) and \( \mathcal{F}[-] \) are evaluation contexts satisfying \( x : \text{bool} \Rightarrow \mathcal{E}[x] \equiv \mathcal{E}[x] \in \mathcal{F}[x] \).

The above cannot be established as a judgmental equality in most type theories, and while it is well-known how to implement extensional booleans in a proof-theoretic setting, the same techniques cannot be applied to inductive types with recursive generators, such as natural numbers. Function extensionality follows immediately from the definitions of the open judgments and equality at function type.

3 Proof Refinement Logic

It is crucial, after defining a computational type theory from a programming language, to devise a proof theory that is sound and facilitates formalization and computer checking. Proof theories prioritize usability and are often incomplete; as such, we have been experimenting with multiple different designs in parallel with RedPRL to find out what works the best for our intended applications; RedPRL is our first such experiment, but there are many other possibilities within the design space [15].

For example, some proof theories achieve a decision procedure for type checking by including more data in the terms, at the expense of making certain kinds of equational reasoning more baroque; this approach has some undeniable advantages in comparison to RedPRL’s more extrinsic style, and we are currently studying it as a means to alleviate a number of practical difficulties in the RedPRL system. We do not at this time recognize any particular proof theory as the “canonical” one.

The proof theory in RedPRL is a proof (program) refinement logic in the tradition of Nuprl [16]; as such, it is oriented around the decomposition of proof goals into subgoals, and the extraction of computational evidence for main goals from the evidence for the subgoals, a refinement of the LCF
methodology first synthesized by [7]. At a technical level, the RedPRL proof logic is an extraction-oriented sequent calculus with judgments like the following, which asserts that the type $A$ is inhabited under the assumptions $H$, with realizer or witness $M$:

$$H \implies A \text{ true } \rightsqarrow M$$

In the above, $H$ and $A$ are inputs to the judgment, whereas $M$ is an output to the judgment: this means that it does not appear in the statement of the goal, but is instead synthesized by the proof refinement system in the course of the proof. A proof refinement logic for this judgment is a signature of refinement rules, which explain how to decompose one such sequent into a collection of other sequents whose witnesses can be combined into a witness for the main sequent.

In RedPRL, these refinement rules are written downward as in the following example, where $(\ast [x : A] (B x))$ is the programming notation for the dependent pair type $(x : A) \times B$:

$$H \implies (\ast [x : A] (B x)) \text{ true } \rightsqarrow (\text{tuple } [\text{proj1 } M] [\text{proj2 } N])$$

by sigma/intro

\[
\begin{align*}
H &\implies A \text{ true } \rightsqarrow M \\
H &\implies (B \overline{M}) \text{ true } \rightsqarrow N \\
H, x : A &\implies (B x) \text{ type} \text{pre}
\end{align*}
\]

Above, the binding structure of the rule is indicated with arrows; in general, outputs flow downward through the subgoals and upward to the output of the main goal. As can be seen, our schema for refinement rules generalizes the one used in Nuprl, in that it permits the statements of subgoals to depend on the realizers of earlier subgoals [25]; this facility also appears in the latest version of Coq’s refinement engine [24], and is also present in the Matita proof assistant [6].

The complete collection of RedPRL’s refinement rules can be viewed online [26].

Auxiliary subgoals In addition to the two familiar subgoals to the introduction rule for dependent pairs, there is a third subgoal required to establish that $(B x)$ is actually a genuine family of types indexed in $A$. In proof assistants like Agda and Coq, these kinds of obligations are checked automatically; however, RedPRL is based on realizers rather than checkable proof terms, so we must explicitly emit such a subgoal in order to preserve soundness. In nearly all cases, such auxiliary subgoals can be discharged automatically using the auto tactic.

3.1 Rule Composition and Proof Tactics

A proof in RedPRL is built from tactics [18]; every refinement rule is a tactic, but the tactics include composite forms like the following:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1 ; t_2$</td>
<td>“run $t_1$ on the current goal, and then run $t_2$ on the resulting subgoals”</td>
</tr>
<tr>
<td>$t ; [t_0, \ldots, t_n]$</td>
<td>“run $t$ on the current goal, and then run $t_1$ on the $i$th resulting subgoal”</td>
</tr>
<tr>
<td>$t_1</td>
<td>t_2$</td>
</tr>
<tr>
<td>auto</td>
<td>“use automation to decompose the current goal as much as possible”</td>
</tr>
</tbody>
</table>

...
In order to ease the process of constructing proofs and programs in RedPRL, the tactic language is also equipped with special notation for certain frequent combinations of rules and tactics. For instance:

\[
\text{lam} \ x \ y \Rightarrow \begin{cases} 
\text{use} \ x, \\
\text{use} \ y 
\end{cases} \xrightarrow{\text{elaborate}} 
\text{pi/intro;}
\]
\[
\begin{array}{l}
\begin{aligned}
&\text{pi/intro;} \\
&\text{with} \ x \Rightarrow \\
&\text{pi/intro;} \\
&\text{with} \ y \Rightarrow \\
&\text{sigma/intro;} \\
&\text{use} \ x, \\
&\text{use} \ y, \\
&\text{auto}
\end{aligned}
\end{array}
\]
\[
\text{auto}
\]

4 Cubical Type Theory

The theory underlying RedPRL differs from earlier computational type theories (notably Nuprl) by extending the syntax of the programming language with dimension expressions encoding higher-dimensional path structure. A dimension expression \( r \) is either a dimension name \( i \) (representing a generic element of an abstract interval), or a constant 0 or 1 representing one of the two endpoints.

In Cartesian cubical computational type theory, there is always an ambient context \( \Psi \) of dimension names, signifying the dimensions accessible to the program. Dimension names behave like variables because one can substitute a dimension expression for a name, where a 0- or 1-substitution takes the 0- or 1-face, respectively, and substituting one name for another takes the corresponding diagonal.

For example, let \( M \) be a program indexed by dimension \( i \), \( M(0/i) \) is its 0-face, \( M(1/i) \) is its 1-face, and \( M(j/i) \) is the diagonal identifying dimensions \( i \) and \( j \). A program indexed by \( n \) dimension names (i.e., by an \( n \)-fold Cartesian product of abstract intervals) forms an abstract \( n \)-dimensional cube. Using dimension names, it is possible to express higher-dimensional structure at the judgmental level. For example, the judgment \( M \triangleq N \in A[i] \) means that \( M \) and \( N \) are equal lines, or paths, varying in the dimension \( i \). In general, the judgements \( A \triangleq B \text{ type } [\Psi] \) and \( M \triangleq N \in A[\Psi] \) refer to \( |\Psi| \)-dimensional structure varying in the dimensions \( \Psi \). (See Angiuli et al. [3] for a more detailed introduction to cubical type theory.)

4.1 Stable Computation

Dimension substitutions significantly complicate the story of computational type theory, because they do not in general commute with computation. Consider the higher inductive type representing a circle, described in part in Figure 2. The program \( \text{S1-rec(Bool;loop;true;false)} \) evaluates to \( \text{false} \). However, if we substitute 0 for \( i \) before evaluating, the resulting program instead evaluates to \( \text{true} \):

\[
\text{S1-rec(Bool;loop;true;false)} \Rightarrow \text{S1-rec(Bool;base;true;false)} \\
\Rightarrow \text{true}.
\]

This is a serious issue if we hope to close our type theory under a computational notion of equivalence. We therefore must restrict our type theory to only recognize programs for which computation and dimension substitution commute (up to judgmental equality); this essential restriction ensures that
the program above shall not inhabit the \texttt{bool} type. We identify a class of stable computation rules that always commute with dimension substitution, and therefore can be used to simplify programs without first establishing that they are well-typed.

Angiuli et al. [4] identify many computation rules as stable, including all rules present in traditional type theory, as they are defined uniformly across all dimension contexts. \textit{RedPRL} is equipped with a sophisticated algorithm to identify more subtle instances of stable computation, by taking into account which dimensions are bound and thus unaffected by substitutions. However, our experience shows that this algorithm does not help much in practice: many difficult cases that appear in concrete formalization efforts are not stable on the nose, but rather coherent only up to the typed judgmental equality.

On the other hand, programs that are already known to be well-typed admit a much broader collection of computation rules. We believe that by focusing our efforts in the future on developing an ergonomic theory of typed programs, we can unleash a richer version of definitional equivalence including equations not justified by stable computation alone.

### 4.2 Kan Operations and Kinds

In order to perform the homotopy-theoretic constructions available in homotopy type theory, we must equip types with Kan operations that describe how to compose and invert lines, and how to transport elements along lines of types [3,4]. Types equipped with such operations are called Kan, and written \( A \text{ type}_{\text{kan}} [\Psi] \). All \textit{RedPRL} type formers present in homotopy type theory—including dependent functions, dependent pairs, paths, higher inductive types, and type universes—preserve being Kan.

However, \textit{RedPRL} also contains types \textit{not} present in homotopy type theory, notably, exact equality types internalizing judgmental equality. These cannot be equipped with Kan operations, because equality is finer than homotopy and therefore not invariant under paths. Such pretypes are written \( A \text{ type}_{\text{pre}} [\Psi] \).

Pretypes and Kan types are part of a continuum of Kan structure recognized by \textit{RedPRL}, such as the types with only partial Kan structure, or those with stronger properties (such as having no non-trivial paths). We are able to further refine our type theory with a language of (currently, five) kinds \( \kappa \) governing the type equality judgment \( A = B \text{ type}_\kappa [\Psi] \), resulting in a theory more expressive than other two-level type theories such as [5].
5 Related and Future Work

RedPRL is under active development. We are currently implementing support for general higher inductive types [13], which will greatly expand the range of homotopy-theoretic proofs expressible in RedPRL. We are also actively experimenting with alternate implementations of complex Kan operations (in particular, for the $V$ and $F_{\text{com}}$ types in Angiuli et al. [4]).

The use of cubical structure in constructive models of homotopy type theory was pioneered by Bezem et al. [9], and has since been studied extensively by ourselves and others [2–4, 13, 14, 20]. There are many subtle technical choices possible in this arena—such as the tradeoffs between expressivity of the dimension language and Kan operations—but these are outside the scope of this survey article.

With Anders Mörberg, we are also implementing the yacctt type checker for a variant of Cartesian cubical type theory whose formal properties mirror those of cubicaltt, rather than the proof refinement logic of RedPRL. In connection with this project, we are studying the question of normal forms and algorithmic definitional equivalence for Cartesian cubical type theory. Despite these efforts, the natural number $n$ in $\mathbb{Z}/n\mathbb{Z}$ mentioned in the introduction remains to be computed successfully: attempts in cubicaltt have consumed excessive memory and failed to terminate, while RedPRL and yacctt do not yet contain sufficient homotopy-theoretic results.

Acknowledgements

We would like to thank everyone who has contributed to the RedPRL project and its predecessors, including Eugene Akentyev, Tim Baumann, David Thrane Christiansen, Daniel Gratzer, Darin Morrison, Vincent Rahli and James Wilcox.

This research was sponsored by the National Science Foundation under grant number DMS-1638352 and the Air Force Office of Scientific Research under grant number FA9550-15-1-0053. The second author would also like to thank the Isaac Newton Institute for Mathematical Sciences for its support and hospitality during the program “Big Proof” when part of work on this paper was undertaken; the program was supported by Engineering and Physical Sciences Research Council under grant number EP/K032208/1. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, government or any other entity.

References


