The RedPRL Proof Assistant (Invited Paper)

Carlo Angiuli
Carnegie Mellon University
cangiuli@cs.cmu.edu

Evan Cavallo
Carnegie Mellon University
ecavallo@cs.cmu.edu

Kuen-Bang Hou (Favonia)
Institute for Advanced Study
favonia@math.ias.edu

Robert Harper
Carnegie Mellon University
rwh@cs.cmu.edu

Jonathan Sterling
Carnegie Mellon University
jmsterli@cs.cmu.edu

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 3 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.

Submitted to:
LFMTP 2018

$$\begin{array}{ll} \text{Expressions} & M,N,O :\equiv x \mid (x:M)\to N \mid \lambda x.M \mid M\,N \\ & \quad \mid (x:M)\times N \mid \langle M,N\rangle \mid \texttt{fst}(M) \mid \texttt{snd}(M) \\ & \quad \mid \texttt{bool} \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}(M;N;O) \end{array}$$

$$\frac{}{(x:M)\to N\ \texttt{val}} \qquad \frac{}{\lambda x.M\ \texttt{val}} \qquad \frac{M \mapsto M'}{M\,N \mapsto M'\,N} \qquad \frac{}{(\lambda x.M)\,N \mapsto M[N/x]} \qquad \frac{}{(x:M)\times N\ \texttt{val}}$$

$$\frac{}{\langle M,N\rangle\ \texttt{val}} \qquad \frac{M \mapsto M'}{\texttt{fst}(M) \mapsto \texttt{fst}(M')} \qquad \frac{}{\texttt{fst}(\langle M,N\rangle) \mapsto M} \qquad \frac{M \mapsto M'}{\texttt{snd}(M) \mapsto \texttt{snd}(M')}$$

$$\frac{}{\texttt{snd}(\langle M,N\rangle) \mapsto N} \qquad \frac{}{\texttt{bool}\ \texttt{val}} \qquad \frac{}{\texttt{true}\ \texttt{val}} \qquad \frac{}{\texttt{false}\ \texttt{val}} \qquad \frac{M \mapsto M'}{\texttt{if}(M;N;O) \mapsto \texttt{if}(M';N;O)}$$

$$\frac{}{\texttt{if}(\texttt{true};N;O) \mapsto N} \qquad \frac{}{\texttt{if}(\texttt{false};N;O) \mapsto O}$$

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 $M\ \texttt{val}$. 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., $\texttt{bool}$), and which programs each type classifies (e.g., those that evaluate to $\texttt{true}$ or $\texttt{false}$). 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 $A \doteq B\ \texttt{type}$, when $A \Downarrow A_0$ (i.e., $A \mapsto^* A_0$ and $A_0\ \texttt{val}$), $B \Downarrow B_0$, 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 $A \doteq A\ \texttt{type}$, written $M \doteq N \in A$, 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 $A\ \texttt{type}$ when $A \doteq A\ \texttt{type}$, and $M \in A$ when $M \doteq M \in A$.

Concretely, the booleans are defined by the following clauses:

$$\texttt{bool} \doteq \texttt{bool}\ \texttt{type} \qquad\qquad \texttt{true} \doteq \texttt{true} \in \texttt{bool} \qquad\qquad \texttt{false} \doteq \texttt{false} \in \texttt{bool}$$

That is, $\texttt{bool}$ is a canonical type with canonical (unequal) elements $\texttt{true}$ and $\texttt{false}$. The canonicity property follows directly from the above definitions.

Theorem 1 (Canonicity)**.** *For any $M \in \texttt{bool}$, $M \Downarrow \texttt{true}$ or $M \Downarrow \texttt{false}$.*

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

$$\begin{aligned}(x:A) \to B \doteq (x:C) \to D\ \texttt{type} &\iff A \doteq C\ \texttt{type}\ \text{and}\ x:A \gg B \doteq D\ \texttt{type}\\ \lambda x.M \doteq \lambda x.N \in (x:A) \to B &\iff x:A \gg M \doteq N \in B\end{aligned}$$

$$\begin{aligned}(x:A) \times B \doteq (x:C) \times D\ \texttt{type} &\iff A \doteq C\ \texttt{type}\ \text{and}\ x:A \gg B \doteq D\ \texttt{type}\\ \langle M,N\rangle \doteq \langle O,P\rangle \in (x:A) \times B &\iff M \doteq O \in A\ \text{and}\ N[M/x] \doteq P[O/x] \in B[M/x]\end{aligned}$$

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 $\texttt{bool}$ type (for all terms $M \doteq M \in \texttt{bool}$ and $x : \texttt{bool} \gg N \doteq N \in C$):

$$N[M/x] \doteq \texttt{if}(M; N[\texttt{true}/x]; N[\texttt{false}/x]) \in C[M/x]$$

(This equation is called the *Shannon expansion*, a widely used tool for deciding boolean formulae.)

The above cannot be established as a judgmental equality in most type theories, and while it is known how to implement extensional booleans in a proof-theoretic setting (at least in the simply-typed case), the same techniques cannot be applied to inductive types with recursive generators (such as natural numbers), nor even to the empty coproduct if regarded as a positive type.[1] In computational type theory, however, all of these unicity principles, as well as function extensionality, follow immediately from the definitions of the open judgments and equality for each 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

[1]The universal property of the positive version of the empty type entails a collapse of definitional equivalence in an inconsistent context; in a traditional proof-theoretic account of type theory, this disrupts strong normalization.

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:

$$H \Longrightarrow A\ \mathtt{true} \rightsquigarrow M$$

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:

$$\begin{tikzcd}[row sep=small]
H \Longrightarrow (x:A)\times B(x)\ \mathtt{true} \rightsquigarrow \langle & M, & N\rangle \\
\text{by } \textbf{sigma/intro} & & \\
H \Longrightarrow A\ \mathtt{true} \rightsquigarrow & M \arrow[uu, bend right] \arrow[d, bend right] & \\
H \Longrightarrow B(& M)\ \mathtt{true} \rightsquigarrow & N \arrow[uuu, bend right] \\
H, x:A \Longrightarrow B(x)\ \mathtt{type} & &
\end{tikzcd}$$

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:

Expression	*Meaning*
$\mathtt{t}_1;\mathtt{t}_2$	"run $\mathtt{t}_1$ on the current goal, and then run $\mathtt{t}_2$ on the resulting subgoals"
$\mathtt{t};[\mathtt{t}_0,\ldots,\mathtt{t}_\mathtt{n}]$	"run $\mathtt{t}$ on the current goal, and then run $\mathtt{t}_\mathtt{i}$ on the *i*th resulting subgoal"
$\mathtt{t}_1 \mid \mathtt{t}_2$	"try running $\mathtt{t}_1$ on the current goal, but if that fails, run $\mathtt{t}_2$ instead"
auto	"use automation to decompose the current goal as much as possible"
…	

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, the following tactic introduces two variables and pairs them, solving the goal $(x : A) \to (y : B(x)) \to (x : A) \times B(x)$:

$$\begin{array}{l}\textsf{lam}\ x\ y \Rightarrow \\ \quad \left\{\begin{array}{l}\textsf{use}\ x,\\ \textsf{use}\ y\end{array}\right\}\end{array} \xRightarrow{\textit{elaborate}} \begin{array}{l} \textsf{pi/intro}; \\ \left[\begin{array}{l} \textsf{with}\ x \Rightarrow \\ \quad \textsf{pi/intro}; \\ \quad \left[\begin{array}{l} \textsf{with}\ y\Rightarrow \\ \quad \textsf{sigma/intro}; \\ \quad \left[\begin{array}{l}\textsf{use}\ x,\\ \textsf{use}\ y,\\ \textsf{auto}\end{array}\right], \\ \textsf{auto}\end{array}\right], \\ \textsf{auto}\end{array}\right]\end{array}$$

While the tactic notation above is built-in for convenience, users can also define their own tactics and tacticals, though **RedPRL** does not yet provide any facilities for extending the concrete notation.

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 Ψ 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, $M\langle 0/i\rangle$ is its 0-face, $M\langle 1/i\rangle$ is its 1-face, and $M\langle j/i\rangle$ 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 $M \doteq N \in A\ [i]$ means that M and N are equal *lines*, or paths, varying in the dimension i. In general, the judgements $A \doteq B\ \mathtt{type}\ [\Psi]$ and $M \doteq N \in A\ [\Psi]$ refer to $|\Psi|$-dimensional structure varying in the dimensions Ψ. (See Angiuli et al. [3] for a more detailed introduction to cubical type theory.)

$$\begin{array}{rl} \text{Dimension expressions} & r :\equiv 0 \mid 1 \mid i \\ \text{Expressions} & M,N,O,P :\equiv \cdots \mid \texttt{S1} \mid \texttt{base} \mid \texttt{loop}_r \mid \texttt{S1-rec}(x.M;N;O;i.P) \end{array}$$

$$\frac{}{\texttt{S1}\ \texttt{val}} \qquad \frac{}{\texttt{base}\ \texttt{val}} \qquad \frac{}{\texttt{loop}_i\ \texttt{val}} \qquad \frac{}{\texttt{loop}_0 \mapsto \texttt{base}} \qquad \frac{}{\texttt{loop}_1 \mapsto \texttt{base}}$$

$$\frac{M \mapsto M'}{\texttt{S1-rec}(x.C;M;B;i.L) \mapsto \texttt{S1-rec}(x.C;M';B;i.L)} \qquad \frac{}{\texttt{S1-rec}(x.C;\texttt{base};B;i.L) \mapsto B}$$

$$\frac{}{\texttt{S1-rec}(x.C;\texttt{loop}_j;B;i.L) \mapsto L\langle j/i\rangle}$$

Figure 2: A fragment of the circle type, omitting the Kan structure.

4.1 Stable Computation

Dimension substitutions significantly complicate the story of computational type theory, because they do not in general commute with computation.

To see the problem, let's introduce a higher inductive type representing a circle, described in part in Figure 2; the new constructs include:

- $\texttt{S1}$: the circle type.
- $\texttt{base}$: a point constructor in $\texttt{S1}$, representing a distinguished point in the circle.
- $\texttt{loop}_r$: a path constructor parametrized by the dimension expressions r, representing the loop in the circle. The program that $\texttt{loop}_i$ represents is a line along dimension i from $\texttt{base}$ to $\texttt{base}$, which is witnessed by the computation rules $\texttt{loop}_0 \mapsto \texttt{base}$ and $\texttt{loop}_1 \mapsto \texttt{base}$.
- $\texttt{S1-rec}(x.C;M;B;i.L)$: the eliminator of $\texttt{S1}$ by case analysis, where $x.C$ is the motive, M is the target, and B and $i.L$ are the two methods matching the constructors $\texttt{base}$ and $\texttt{loop}_i$.

With the circle type $\texttt{S1}$, we can see how dimensions might complicate the story of computation. Consider the program $\texttt{S1-rec}(\_.\texttt{bool};\texttt{loop}_i;\texttt{true};\_.\texttt{false})$. It should evaluate to $\texttt{false}$ according to the rules in Figure 2. However, if we substitute 0 for i before evaluating, the resulting program instead evaluates to $\texttt{true}$:

$$\begin{aligned} \texttt{S1-rec}(\_.\texttt{bool};\texttt{loop}_0;\texttt{true};\_.\texttt{false}) &\mapsto \texttt{S1-rec}(\_.\texttt{bool};\texttt{base};\texttt{true};\_.\texttt{false}) \\ &\mapsto \texttt{true}. \end{aligned}$$

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 also 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: all rules present in traditional type theory, as they are defined uniformly across all dimension contexts; moreover, rules such as $\texttt{loop}_0 \mapsto \texttt{base}$ will never be affected by substitutions. **RedPRL** is equipped with an 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 as much in practice as one might hope: many difficult cases that arise in concrete formalization efforts are not precisely stable, rather only up to 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 employ 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 A equipped with such operations are called Kan, and written $A\ \mathtt{type}_{\mathtt{kan}}\ [\Psi]$. They are also called *fibrant types* because the fibrancy in many models of interest corresponds to implementing Kan operations. All **RedPRL** type formers present in homotopy type theory—including dependent functions, dependent pairs, paths, higher inductive types, and type universes—preserve being Kan.

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

Pretypes and Kan types are part of a continuum of Kan structures recognized by **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* κ governing the type equality judgment $A \doteq B\ \mathtt{type}_{\kappa}\ [\Psi]$, resulting in a theory more expressive than other two-level type theories such as [5]. Equality types of a type with no non-trivial paths can be made Kan, for example.

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**. One consequence would be the ability to fully model homotopy type theory and various two-level type theories[2]; see [13]. We are also actively experimenting with alternate implementations of complex Kan operations (in particular, for the $\mathtt{V}$ and $\mathtt{Fcom}$ types in Angiuli et al. [4]).

The use of the 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 Mörtberg, 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

[2]Except the resizing rules in the homotopy type system proposed by Voevodsky [30].

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, Anders Mörtberg, 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

[1] Mathieu Anel, Georg Biedermann, Eric Finster & André Joyal (2017): *A Generalized Blakers-Massey Theorem*. Available at `https://arxiv.org/abs/1703.09050`.

[2] Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Kuen-Bang Hou (Favonia), Robert Harper & Daniel R. Licata (2017): *Cartesian Cubical Type Theory*. Available at `https://github.com/dlicata335/cart-cube`. Preprint.

[3] Carlo Angiuli, Robert Harper & Todd Wilson (2017): *Computational Higher-Dimensional Type Theory*. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, ACM, New York, NY, USA, pp. 680–693, doi:10.1145/3009837.3009861.

[4] Carlo Angiuli, Kuen-Bang Hou (Favonia) & Robert Harper (2017): *Computational Higher Type Theory III: Univalent Universes and Exact Equality*. Available at `https://arxiv.org/abs/1712.01800`.

[5] Danil Annenkov, Paolo Capriotti & Nicolai Kraus (2017): *Two-Level Type Theory and Applications*. Available at `https://arxiv.org/abs/1705.03307`.

[6] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen & Enrico Tassi (2011): *The Matita Interactive Theorem Prover*. In: *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, pp. 64–69, doi:10.1007/978-3-642-22438-6_7.

[7] Joseph L. Bates (1981): *A Logic for Correct Program Development*. Technical Report TR81-455, Cornell University. Available at `http://hdl.handle.net/1813/6295`. Revision of Cornell University Ph.D. thesis submitted August 1979.

[8] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau & Bas Spitters (2017): *The HoTT Library: A Formalization of Homotopy Type Theory in Coq*. In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, ACM, New York, NY, USA, pp. 164–172, doi:10.1145/3018610.3018615.

[9] Marc Bezem, Thierry Coquand & Simon Huber (2014): *A model of type theory in cubical sets*. In: *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, 26, Dagstuhl Publishing, Toulouse, France, pp. 107–128, doi:10.4230/LIPIcs.TYPES.2013.107.

[10] Edwin Brady (2013): *Idris, a general-purpose dependently typed programming language: Design and implementation.* Journal of Functional Programming 23(5), pp. 552–593, doi:10.1017/S095679681300018X.

[11] Guillaume Brunerie (2016): *On the homotopy groups of spheres in homotopy type theory.* Ph.D. thesis, Universit Nice Sophia Antipolis. Available at `https://arxiv.org/abs/1606.05916`.

[12] Guillaume Brunerie, Kuen-Bang Hou (Favonia), Evan Cavallo, Eric Finster, Jesper Cockx, Christian Sattler, Chris Jeris, Michael Shulman et al. (2018): *Homotopy Type Theory in Agda.* Available at `https://github.com/HoTT/HoTT-Agda`.

[13] Evan Cavallo & Robert Harper (2018): *Computational Higher Type Theory IV: Inductive Types.* Available at `https://arxiv.org/abs/1801.01568`.

[14] Cyril Cohen, Thierry Coquand, Simon Huber & Anders Mörtberg (2018): *Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom.* In: 21st International Conference on Types for Proofs and Programs (TYPES 2015), 69, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 5:1–5:34, doi:10.4230/LIPIcs.TYPES.2015.5.

[15] Cyril Cohen, Thierry Coquand, Simon Huber & Anders Mörtberg (2018): *`cubicaltt`: Experimental implementation of Cubical Type Theory.* `https://github.com/mortberg/cubicaltt`.

[16] Robert L. Constable, et al. (1985): *Implementing Mathematics with the Nuprl Proof Development Environment.* Prentice-Hall, Englewood Cliffs, NJ. Available at `http://www.nuprl.org/book/`.

[17] Floris van Doorn, Jakob von Raumer & Ulrik Buchholtz (2017): *Homotopy Type Theory in Lean.* In: Interactive Theorem Proving, Springer, Cham, pp. 479–495, doi:10.1007/978-3-319-66107-0_30.

[18] Michael Gordon, Robin Milner & Christopher Wadsworth (1979): *Edinburgh LCF: A Mechanized Logic of Computation.* Lecture Notes in Computer Science 78, Springer-Verlag, Berlin, Heidelberg, doi:10.1007/3-540-09724-4.

[19] Chris Kapulkin & Peter LeFanu Lumsdaine (2016): *The Simplicial Model of Univalent Foundations (after Voevodsky).* Available at `https://arxiv.org/abs/1211.2851`.

[20] Daniel R. Licata & Guillaume Brunerie (2014): *A cubical type theory.* Available at `http://dlicata.web.wesleyan.edu/pubs/lb14cubical/lb14cubes-oxford.pdf`. Talk at Oxford Homotopy Type Theory Workshop.

[21] Peter LeFanu Lumsdaine & Mike Shulman (2017): *Semantics of higher inductive types.* Available at `https://arxiv.org/abs/1705.07088`.

[22] P. Martin-Löf (1984): *Constructive Mathematics and Computer Programming.* Philosophical Transactions of the Royal Society of London Series A 312, pp. 501–518, doi:10.1098/rsta.1984.0073.

[23] Conor McBride (2005): *Epigram: Practical Programming with Dependent Types.* In: Proceedings of the 5th International Conference on Advanced Functional Programming, AFP'04, Springer-Verlag, Berlin, Heidelberg, pp. 130–170, doi:10.1007/11546382_3.

[24] Arnaud Spiwack (2011): *Verified Computing in Homological Algebra, A Journey Exploring the Power and Limits of Dependent Type Theory.* Ph.D. thesis, École Polytechnique. Available at `https://pastel.archives-ouvertes.fr/pastel-00605836`.

[25] Jonathan Sterling & Robert Harper (2017): *Algebraic Foundations of Proof Refinement.* Available at `https://arxiv.org/abs/1703.05215`.

[26] The RedPRL Development Team (2018): *RedPRL – the People's Refinement Logic.* Available at `http://www.redprl.org/`.

[27] The Univalent Foundations Program (2013): *Homotopy Type Theory: Univalent Foundations of Mathematics.* `http://homotopytypetheory.org/book`, Institute for Advanced Study.

[28] Vladimir Voevodsky (2010): *The equivalence axiom and univalent models of type theory.* Available at `http://www.math.ias.edu/vladimir/files/CMU_talk.pdf`. Notes from a talk at Carnegie Mellon University.

[29] Vladimir Voevodsky (2010): *Univalent Foundations Project*. Available at `http://www.math.ias.edu/vladimir/files/univalent_foundations_project.pdf`. Modified version of an NSF grant application.

[30] Vladimir Voevodsky (2013): *A type system with two kinds of identity types*. Available at `https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/HTS.pdf`. Slides available at `https://uf-ias-2012.wikispaces.com/file/view/HTS_slides.pdf/410105196/HTS_slides.pdf`.