Theorem prover Arend

Fedor Part Valery Isaev Sergey Sinchuk

Abstract

This paper is an exposition of the theorem prover Arend developed at JetBrains Research. Arend language has been designed to facilitate the formalization of mathematics within the framework of univalent foundations and with the focus on constructive mathematics. It is based on a variant of homotopy type theory with cubical syntax that is more closely aligned with Martin-Löf Type Theory (MLTT) than fully computational cubical type theories, while still retaining some of their computational properties useful in practice. Among the most notable features of the Arend language is its powerful system of classes and records supporting subsumptive subtyping, partial implementations and anonymous extensions, which provides extensive flexibility for constructing hierarchies of definitions. Formalization in Arend is streamlined by the IntelliJ Arend plugin, which transforms IntelliJ IDEA into a full-fledged integrated development environment (IDE) for the Arend language. The main Arend library arend-lib includes formalizations in constructive algebra, topology and synthetic homotopy theory.

1 Introduction

Interactive theorem provers (ITPs) are software tools designed to assist in the creation, verification, and exploration of formal proofs by providing a rigorous framework for encoding and checking logical reasoning. They leverage formal systems, such as type theory or higher-order logic, to provide an interactive environment where users can encode theorems, construct proofs, and validate their correctness with absolute precision. By eliminating human error, ITPs have become invaluable not only in formalizing complex mathematical results but also in verifying the correctness of critical software and hardware systems, bridging the gap between pure mathematics and practical applications. Modern ITPs, such as Coq, Agda, Lean, and HOL-based systems like Isabelle and HOL4, enable a wide range of use cases, from verifying algorithms and protocols to formalizing groundbreaking theorems such as the Four-Color Theorem and the Feit-Thompson Theorem.

Central to many ITPs is dependent type theory, a formalism that unifies programs and proofs within a single framework. Systems like Coq, Lean and Agda, built upon the Calculus of Inductive Constructions (CIC) or Martin-Löf Type Theory (MLTT), exemplify the power of dependent type theory in encoding rich mathematical structures and reasoning about them formally. Whereas Coq and Lean (based on CIC) do not particularly favor constructivism (Lean even has the axioms of choice and excluded middle built-in), Agda (based on MLTT) leans more towards constructive mathematics. HOL-based systems, rooted in classical higher-order logic, offer an alternative approach that is more familiar to traditional mathematicians and computer scientists. However, dependent type theories surpass HOL in expressiveness by allowing types to depend on values, enabling constructive reasoning with computational interpretations via term reductions, and unifying proofs and programs within a single framework, capabilities that HOL's classical framework lacks.

In the early 2000s, the fields of formalized mathematics and dependently typed languages — previously limited mostly to communities of computer scientists and logicians — began to draw significant attention from pure mathematicians. A pivotal moment in this shift was the formulation of *Univalent Foundations* (UF) of mathematics by Vladimir Voevodsky between 2006 and 2009. This new approach, inspired by the homotopy semantics of MLTT, interprets types as homotopy types[1] rather than as sets. In particular, the

[1]The canonical geometric example of a homotopy type is a topological space up to weak homotopy equivalence. Other combinatorial examples include presheaves such as simplicial or cubical sets up to a suitable equivalence.

equality type $a =_A b$ is interpreted as the type of paths between a and b in A, for $p, q : a =_A b$, the type $p =_{a=_A b} q$ is the type of homotopies between p and q in A, and so on. The homotopy semantics suggested that MLTT can serve as a basis for a formal system where the notion of a homotopy type is taken as primitive instead of the notion of a set. This led to formulation of Homotopy Type Theory (HoTT) as an extension of MLTT with *univalence axiom* $(A \cong B) \cong (A = B)$, which allows to identify two equivalent types, and *higher inductive types*, a certain generalization of inductive types allowing higher path constructors.

The impact of HoTT/UF on formalization of mathematics is two-fold:

1. Many standard constructions and theorems in homotopy theory can now be elegantly formalized using HoTT, where homotopy types serve as primitives rather than sets. This method, known as *synthetic homotopy theory*, utilizes the equality type to encapsulate complex homotopy structures. Set-theoretic versions of these formalizations tend to be more cumbersome. This synthetic approach not only simplifies the formalization of existing theories but also has the potential to reshape many areas of modern mathematics by adopting a higher categorical perspective over traditional set-theoretic approaches.

2. HoTT/UF provides a framework for set-theoretic formalization where sets are homotopy types with trivial homotopy structure and propositions are sets with at most one element. One can do formalizations completely confined to this set theory fragment of HoTT. As such a framework HoTT/UF augments MLTT with a fair degree of *extensionality*:

    (a) Quotients can be easily defined in HoTT using higher inductive types. Meanwhile quotients are notoriously hard to deal with in pure MLTT due to intensionality of equality.

    (b) Univalence axiom implies propositional and function extensionality: any two equivalent propositions are equal and any two pointwise equal functions are equal.

    (c) Univalence axiom implies *structure identity principle*: isomorphic structures (such as groups, topological spaces and so on) are equal.

This approach can, in principle, be implemented using an ITP based on MLTT, such as Coq or Agda[2], by merely appending univalence and higher inductive types as axioms devoid of any computational content. However, this severely impairs the computational properties of MLTT, which are crucial for practical formalization efforts. Additionally, there are several other problematic issues stemming from the fact that these ITPs were not originally designed with native support for HoTT/UF. For example, the universe of propositions Prop in Coq or Agda does not correspond to the type of propositions as defined in HoTT/UF.

Theorem prover Arend. The development of Arend began in 2015 at JetBrains Research with the goal of creating a modern ITP featuring *native* support for HoTT/UF. Its two principal components are the Arend language, which is based on a variant of MLTT specifically tailored to integrate seamlessly with HoTT/UF, and a suite of tooling provided by the Arend plugin for IntelliJ IDEA. The key components of the Arend ITP are:

1. **Type theory.** At the core of the type theory of Arend is the *homotopy type theory with interval type* (HoTT-I), formulated by Valery Isaev in 2014. It consists of two components:

    (a) **Variant of MLTT with the interval type:** a minor, more extensional modification of MLTT that incorporates a primitive type $\mathbf{I}$ for the interval. The equality type (or path type) is defined in terms of $\mathbf{I}$. Function extensionality is derivable and is fully computational in this theory. This setup gives rise to a variant of cubical syntax: n-dimensional homotopies in a type A are naturally represented in HoTT-I as functions $\mathbf{I}^n \to A$.

    (b) **Univalence axiom:** a form of univalence axiom, which is a built-in axiom with some computational rules.

[2]An important caveat is that the ITP should not incorporate built-in features that contradict the univalence axiom. For instance, Lean is unsuitable for UF as it incorporates a strong form of the axiom of choice built-in.

The native support for HoTT/UF in the type theory of Arend is further amplified by the following extensions of HoTT-I:

(a) **Inductive types with conditions.** Using the primitive $\mathbf{I}$, a path constructor can be defined simply as a constructor $\mathsf{pcon}\,(i : \mathbf{I})$ with an interval type parameter, or multiple parameters for higher-dimensional homotopies. The only difference with an ordinary constructor is that there could be *conditions* that dictate how pcon evaluates at the ends of the interval to other constructors: $\mathsf{pcon\,left} \Rightarrow \mathsf{con}_1$, $\mathsf{pcon\,right} \Rightarrow \mathsf{con}_2$. In particular, it implies $con_1 = con_2$ but does not lead to contradictions as in the case of standard inductive types in MLTT, thanks to a modified elimination principle that requires the preservation of conditions. This provides support for higher inductive types and, in particular, quotients with computational behavior.

(b) **Universes Prop and Set matching h-propositions and h-sets.** Arend incorporates an *impredicative* universe Prop that captures the logic of h-propositions and a predicative hierarchy of universes $\mathsf{Set}\,0 \subset \mathsf{Set}\,1 \subset \dots$ for h-sets. The universes Prop and Set provide a *syntactic* framework that facilitates working within the set theory domain, akin to the elementary topos of h-sets, and distinctively delineate the set-level aspect of the type theory from the higher levels, ensuring that set-theoretic formalization is confined strictly to the subtheory prescribed by these universes.

(c) **Polymorphism for homotopy levels.** Arend introduces a mechanism for polymorphism based on homotopy level. This allows a single polymorphic definition to be instantiated across universes of varying homotopy level, including Prop, $\mathsf{Set}\,n$, and the universes of all types $\mathsf{Type}\,n$. This feature enables greater flexibility and reusability of definitions.

The above outlines the core fragment of Arend's type theory as it relates to HoTT/UF. Its key properties include:

(a) **Computational behavior.** Unlike MLTT augmented merely with axioms for univalence and higher inductive types, Arend's type theory includes a number of computational reduction rules for involved terms, which prove crucial for practical usage. However, it lacks sufficient computational rules to render the theory fully computational: the MLTT property that every closed term of type Nat evaluates to a canonical number does not hold in Arend. The core theory of Arend represents one of the earliest and simplest forms of *cubical type theory*, in which the interval still functions as a type. More advanced, fully computational cubical type theories have since evolved, in which univalence is derivable. However, these approaches are considerably more complex, featuring two-level theories where the interval $\mathbf{I}$ is *not a type*. While the development of such fully computational theories is of theoretical interest, the complexity of two-level frameworks do not currently justify their use in practical applications. The type theory of Arend could potentially be adjusted to enhance its computational aspects, but practical experiences in formalization have not demonstrated a significant need for such enhancements.

(b) **Constructivity.** The type theory of Arend is inherently constructive. In its core theory, the law of excluded middle does not hold, and the axiom of choice is valid only in the form of *unique choice*. While the axioms of classical logic can be consistently added and utilized in libraries, the main Arend library, arend-lib, is dedicated to constructive mathematics, and no standard metas utilize any classical axioms.

Finally, the type theory of Arend includes the following extensions, which are of significant practical importance for formalization and yet are not present in Coq, Lean, or Agda:

(a) **Records with partial implementations and anonymous extensions.** In Arend, type classes and records support multiple inheritance with *partial implementations*. Specifically, if record D extends record C, it is possible to implement an arbitrary subset of C's fields within D. For instance, for a field f of C, its extension D may include an expression $f \Rightarrow a$ that specifies a value a for f. This flexibility blurs the traditional distinction between parameters and fields in classes and records, enabling more versatile hierarchical structure development. Additionally, Arend allows for the creation of records and classes on-the-fly as an *anonymous extension*, denoted as $C\{f_1 \Rightarrow a_1, \ldots, f_k \Rightarrow a_k\}$. This feature facilitates dynamic customization of types without the need to define fully specified new classes or records, thus enhancing modularity and reusability.

   (b) **Array types.** Arend introduces a specialized type for arrays, denoted as $\mathsf{Array}\, A$, where $A : \mathsf{Type}$. This type encompasses several concepts including the type of lists of elements of type A, the type of vectors of elements of type A of fixed length n and the type $\mathsf{Fin}\, n \to A$ of functions from a finite set of cardinality n to A. The type Array is a record with fields $A : \mathsf{Type}$, $\mathsf{len} : \mathsf{Nat}$ and $\mathsf{at} : \mathsf{Fin}\,\mathsf{len} \to A$. The anonymous extension $\mathsf{Array}\, A$ describes arrays of various lengths with elements of type A, and the extension $\mathsf{Array}\, A\, n$ specifies arrays of a fixed length n. A crucial feature of the array type in Arend is its computational extensionality: if two arrays of equal length are elementwise computationally identical, then they are considered computationally equivalent as entire structures.

2. **Proof terms.** In Arend, proof terms do not need to be fully detailed. Users can take advantage of the "inference of implicit arguments" mechanism, which allows them to omit some arguments in expressions (more concretely, the ones that can be inferred via the "inference of implicit arguments mechanism" built into the Arend type checker). Additionally, users can employ metas such as rewrite to construct particularly complicated parts of expressions. Metas should be seen as "expression-level tactics", meaning they can be seamlessly interwoven with standard Arend term constructs. In other words, unlike Coq, Arend does not have separate "proof" and "term" levels – everything (including proofs) in Arend happens on the "term" level.

3. **IntelliJ IDEA plugin.** The easiest way to use Arend is through the tooling provided by the Arend plugin for IntelliJ IDEA. IntelliJ IDEA, developed by JetBrains, is a versatile IDE with intelligent code assistance, robust refactoring tools, and seamless integration with languages like Java, Kotlin, and Python. It has a variety of built-in tools for version control, debugging and testing.

   Arend plugin supports the following functionality:

   (a) **Incremental type checking.** Under normal circumstances the Arend plugin uses the so-called "on-the-fly" (or smart) mode of type checking. In this mode, the type checker operates in a background thread, automatically reprocessing each code modification made by the user.

   (b) **Arend Messages tool window.** The Arend Messages tool window serves as the primary interactive element in Arend, providing essential insight into proof goals, expected types, and error notifications. While writing terms the user can insert hole expressions `{?}` for arbitrary subterms, which would introduce goals shown in Arend Messages. This allows for a convenient way of writing proof terms by gradually breaking down the original proof goal into several subgoals.

   (c) **Editor features.** The Arend editor supports a variety of features that greatly simplify working with Arend code such as: auto-completion of identifiers; quick fixes for Arend errors; intention actions such as adding missing clauses in pattern matching; refactorings such as handling consequences of changing signatures of definitions or moving definitions; navigation tools such as Go to Declaration or Find Usages; quick documentation popups supporting LaTex; parameter hint tooltip which can be invoked in application expressions.

   (d) **Debugger for metas.**

The library arend-lib. The main Arend library arend-lib can be divided into three parts which are developed constructively (without the excluded middle or the axiom of choice):

1. **Constructive mathematics.** This is the main part of the library. The mathematical landscape in the constructive setting is richer than in the classical setting since classically equivalent definitions often become inequivalent constructively.

This part contains the following: schemes via locally ringed locales; PID domains and the proof that they are 1-dimensional Smith domains; splitting fields of polynomials and algebraic closure for countable, decidable fields; connection between zero-dimensional and integral extensions; matrices over commutative rings, determinants, characteristic polynomials, Cayley-Hamilton theorem; linear algebra over Smith domains; integral ring extensions; polynomials over one or several variables; Nakayama's lemma; derivative over topological rings; directed limits for sequences and functions; series and power series; natural, integer, rational, real and complex numbers and various structures on them; categories, functors, adjoint functors, Kan extensions, (co)limits; elementary topoi and Grothendieck topoi; topological spaces, locales, uniform spaces, completion of spaces.

The main references are the books [22, 19] and the paper on constructive complete spaces by Isaev [13].

2. **Synthetic homotopy theory.** The following has been formalized synthetically that is under types as homotopy types viewpoint: Eckmann-Hilton argument; $K_1(G)$; Hopf fibration; localization of universes and modalities; Generalized Blakers-Massey theorem.

3. **Computer science.** Currently this part consists of formalization of high-order term rewriting systems. The planned future formalizations include fragments of computational complexity theory.

Future directions. So far, developments in synthetic homotopy theory in HoTT have been limited. One reason is that types in HoTT are necessarily ∞-groupoids, a special kind of ∞-categories where all morphisms are invertible. This is because equality is symmetric. Attempts to define more general ∞-categorical structures in terms of ∞-groupoids in HoTT have led to the still-open question of definability of (semi-)simplicial types in HoTT[16]. Recently, proposals for *directed* HoTT [8, 7], where types are ∞-categories, have emerged. Future versions of Arend are expected to support a variant of directed HoTT.

The structure of this paper. Section 2 describes the type theory of Arend and compares it to MLTT and "textbook" HoTT [28] in the contexts of set-theoretic formalization and synthetic homotopy theory. Section 3 presents a few examples of Arend code highlighting its novelties – namely, the system of universes, type classes / records, inductive types with conditions and arrays.

2 Underlying type theory of Arend

The type theory of Arend is an extension of the *homotopy type theory with interval type* (HoTT-I). It is a simple modification of Martin-Lof type theory (MLTT) which allows for native support of homotopy type theory and has certain important advantages over MLTT even for formalisations unrelated to homotopy theory.

2.1 Homotopy type theory with interval type

1. **Interval type I**: a new primitive which serves as a basis for the definition of equality type $x =_A y$. Type **I** is defined as a data type with two constructors $\mathsf{left} : \mathbf{I}$, $\mathsf{right} : \mathbf{I}$ and the elimination principle given by the function coe:

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathbf{I}} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathsf{left} : \mathbf{I}} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathsf{right} : \mathbf{I}}$$

$$\frac{\Gamma, x : \mathbf{I} \vdash A \qquad \Gamma \vdash a : A[x := \mathsf{left}] \qquad \Gamma \vdash i : \mathbf{I}}{\Gamma \vdash \mathsf{coe}(\lambda x.A, a, i) : A[x := i]}$$

$$\mathsf{coe}(\lambda x.A, a, \mathsf{left}) \equiv a, \quad \mathsf{coe}(\lambda x.A, a, i) \equiv a \text{ if } x \notin \mathsf{FV}(A)$$

Functions from $\mathbf{I}$ to a type A can be interpreted as paths in A. In HoTT-I, the equality type is defined as the type of paths $\mathsf{Path}(\lambda x.A, a, a')$, which are maps f from $\mathbf{I}$ to A with fixed endpoints $f(\mathsf{left}) \equiv a$ and $f(\mathsf{right}) \equiv a'$:

$$\frac{\Gamma, x : \mathbf{I} \vdash A \qquad \Gamma \vdash a : A[x := \mathsf{left}] \qquad \Gamma \vdash a' : A[x := \mathsf{right}]}{\Gamma \vdash \mathsf{Path}(\lambda x.A, a, a')}$$

$$\frac{\Gamma, x : \mathbf{I} \vdash a : A}{\Gamma \vdash \mathsf{path}(\lambda x.a) : \mathsf{Path}(\lambda x.A, a[x := \mathsf{left}], a[x := \mathsf{right}])}$$

$$\frac{\Gamma \vdash p : \mathsf{Path}(\lambda x.A, a, a') \qquad \Gamma \vdash i : \mathbf{I}}{\Gamma \vdash p \mathbin{@_{a,a'}} i : A[x := i]}$$

$$\begin{gathered}\mathsf{path}(\lambda x.t) \mathbin{@_{a,a'}} i \equiv t[x := i], \quad \mathsf{path}(\lambda x.p \mathbin{@} x) \equiv p \text{ if } x \notin \mathsf{FV}(p) \\ p \mathbin{@_{a,a'}} \mathsf{left} \equiv a, \quad p \mathbin{@_{a,a'}} \mathsf{right} \equiv a'\end{gathered}$$

The standard J-eliminator for equality types in MLTT can be derived from these rules.

2. **Univalence axiom:** the function iso witnessing the principle "isomorphic types are equal" which implies the **structure identity principle** (see 2.3.3 below). Given types A, B, mutually inverse functions f, g and proofs p, q of $f \circ g = \mathsf{id}$ and $g \circ f = \mathsf{id}$, the function iso constructs a family of types $\mathsf{iso}(A, B, f, g, p, q, i)$ parameterized by interval $i : \mathbf{I}$, such that $\mathsf{iso}(A, B, f, g, p, q, \mathsf{left}) \equiv A$ and $\mathsf{iso}(A, B, f, g, p, q, \mathsf{right}) \equiv B$:

$$\frac{\begin{matrix}\Gamma \vdash A & \Gamma, x : A \vdash b : B & \Gamma, x : A \vdash p : a[y := b] = x & \\ \Gamma \vdash B & \Gamma, y : B \vdash a : A & \Gamma, y : B \vdash q : b[x := a] = y & \Gamma \vdash i : \mathbf{I}\end{matrix}}{\Gamma \vdash \mathsf{iso}(A, B, \lambda x.b, \lambda y.a, \lambda x.p, \lambda y.q, i)}$$

$$\begin{gathered}\mathsf{iso}(A, B, \lambda x.b, \lambda y.a, \lambda x.p, \lambda y.q, \mathsf{left}) \equiv A, \quad \mathsf{iso}(A, B, \lambda x.b, \lambda y.a, \lambda x.p, \lambda y.q, \mathsf{right}) \equiv B \\ \mathsf{coe}(\lambda i.\mathsf{iso}(A, B, \lambda x.b, \lambda y.a, \lambda x.p, \lambda y.q, i), a_0, \mathsf{right}) \equiv b[x := a_0] \text{ if } i \notin \mathsf{FV}(A\ B\ b\ a\ p\ q)\end{gathered}$$

2.2 Extensions of HoTT-I in Arend

The core type theory of Arend includes a number of extensions of HoTT-I:

1. **Inductive types with conditions.** Definitions of inductive types in the core type theory of Arend extend the standard definitions with *conditions* on constructors. Conditions allow to glue different constructors when evaluated at certain values of parameters. For example, for the type Int of integers with two constructors $\mathsf{pos}\,(n : \mathsf{Nat})$ and $\mathsf{neg}\,(n : \mathsf{Nat})$ a condition might be $\mathsf{pos}\,0 \equiv \mathsf{neg}\,0$. In general a condition on a constructor $\mathsf{con}\,(a_1 : A_1) \ldots (a_n : A_n)$ is almost the same as a definition of a partial function by pattern matching which specifies how con evaluates. The most interesting examples of conditions involve pattern matching on the interval (such pattern matching is not allowed for functions): it can be used to glue arbitrary constructors con_1 and con_2 by means of the constructor $\mathsf{con}_=\,(i : \mathbf{I})$ with conditions $\mathsf{con}_=\ \mathsf{left} \equiv \mathsf{con}_1$ and $\mathsf{con}_=\ \mathsf{right} \equiv \mathsf{con}_2$. This allows to define quotient types and higher inductive types (see 2.3.2 and 2.4 below).

   The mechanism of inductive types with conditions in Arend has been analised in the paper [31].

2. **Propositions and sets.** Type theory of Arend contains the type Prop – an impredicative universe of all propositions, and a predicative hierarchy of universes of sets $\mathsf{Set}\ 0 \subseteq \mathsf{Set}\ 1 \subseteq \ldots$. Universes Prop and $\mathsf{Set}\ n$ constitute a fragment of the type theory which represents a variant of set theory and provide a framework for conventional set-theoretic formalization.

   The following properties characterize the logic and the universe Prop:

   (a) The operations $\forall$, $\wedge$, $\to$, $\neg$ as well as the constants True, False are defined as usual in propositions-as-types logic, corresponding to $\prod$, $\times$, $\to$, $\cdot \to \mathbf{0}$ and the types $\mathbf{1}$, $\mathbf{0}$, respectively. Operations $\exists$, $\vee$ can be defined as inductive types or via $\sum$, $+$ as $\exists(x : A)P(x) :=\mathsf{nonempty}(\sum_{(x:A)} P(x))$ and $P \vee Q := \mathsf{nonempty}(P + Q)$, where $\mathsf{nonempty}(A)$ is an inductive type (see $(2e)$ below).

   (b) Prop is **impredicative** and the logic of Prop is by default **constructive**: the law of excluded middle $\mathsf{LEM} := \forall(P : \mathsf{Prop})\, P \vee \neg P$ and general axiom of choice do not hold. Propositions are sets: $\mathsf{Prop} \subseteq \mathsf{Set}\ 0$ and $\mathsf{Prop} : \mathsf{Set}\ 0$.

   (c) Propositions $P : \mathsf{Prop}$ satisfy **propositional proof irrelevance** $\mathsf{PI}(P) :=$ $\prod_{(p,q:P)} p = q$. This is one of the main properties that distinguish propositions from other types: a proposition conveys no more information than a truth value.

   (d) **Propositional extensionality** holds for Prop: if $P \leftrightarrow Q$ for $P, Q : \mathsf{Prop}$ then $P = Q$. This is a consequence of proof irrelevance and the univalence axiom.

   (e) The universe Prop defines the syntactic notion of proposition: propositions are types in Prop and the main property of types in Prop is proof irrelevance. But there is also a more semantic notion of an **h-proposition**, which is any type A satisfying proof irrelevance, namely such that $\mathsf{isProp}(A) := \mathsf{PI}(A)$ is provable.

   In the type theory of Arend any definition of an inductive type $P : \mathsf{Type}$, which is an h-proposition, can be supplemented with a proof $p : \mathsf{isProp}(P)$ in which case P gets placed in Prop and becomes a proposition syntactically. This applies also to record types and there are analogous extensions for function definitions which ensure, for example, that $\mathsf{isProp}(A)$ is in Prop. This mechanism allows to define the predicate $\mathsf{nonempty}(A) : \mathsf{Prop}$[3] as inductive type with constructors $\mathsf{in}\,(a : A)$ and $\mathsf{trunc}\,(a\, b : \mathsf{nonempty}(A))\,(i : \mathbf{I})$ with conditions $\mathsf{trunc}\, a\, b\, \mathsf{left} \equiv a$, $\mathsf{trunc}\, a\, b\, \mathsf{right} \equiv b$ and an obvious proof $\lambda x.\lambda y.\mathsf{path}(\mathsf{trunc}\, x\, y) : \mathsf{isProp}(\mathsf{nonempty}(A))$.

   Another key mechanism: for an inductive type P in Prop and an h-proposition A functions $P \to A$ can be defined by simplified elimination where constructors containing parameters of type $\mathbf{I}$ are omitted. This implies $\mathsf{PI}(P)$ for any $P : \mathsf{Prop}$: with this elimination one can construct a function $\mathsf{nonempty}(P) \to P$ which is inverse to the obvious function $P \to \mathsf{nonempty}(P)$ and prove that they are mutually inverse.

   As an alternative to the mechanism for placement of inductive types $P : \mathsf{Type}$ with $p : \mathsf{isProp}(P)$ in Prop Arend supports a mechanism for marking inductive definitions as **truncated definitions** in Prop and limiting definitions of functions $P \to A$ by elimination only to h-propositions A. For example, propositions $\exists(x : A)P(x) : \mathsf{Prop}$ and $P \vee Q : \mathsf{Prop}$ can be defined directly as truncated inductive types rather than via $\mathsf{nonempty}$. We call **h-prop elimination** this kind of elimination from truncated inductive types in Prop. In case A is not a proposition, eliminator includes a proof $p : \mathsf{isProp}(A)$.

   In the presence of univalence elimination from truncated definitions in Prop cannot be extended to all types. Full, unrestricted elimination from propositions to arbitrary types would imply the existence of a global choice operator $\mathsf{gc} : \Pi_{A:\mathsf{Set}\, n}\mathsf{nonempty}(A) \to A$, which is inconsistent with the univalence axiom ([24], Corollary 17.5.3). Moreover propositional proof irrelevance for $P : \mathsf{Prop}$ cannot be strengthened to the computational one $p \equiv q$, $p, q \in P$. Although it is probably consistent, it causes some issues in a type theory with h-prop elimination.

[3] $\mathsf{nonempty}(A)$ is also called *propositional truncation* and denoted $\|A\|_{-1}$

From the above it follows that the universe Prop in Arend can be seen as an extension of the universe Prop in Coq, Lean or Agda with a more general form of elimination for inductive types. In Coq, Lean or Agda elimination from Prop is confined to the universe Prop itself (and is called *small elimination*). In Arend the elimination is more general: it is at least h-prop elimination (for truncated definitions) and sometimes unrestricted elimination (for definitions P supplemented with $p : \mathsf{isProp}(P)$). This aligns with the HoTT approach to logic, which is based on h-propositions.

See Section 3 for examples in Arend language.

(f) As a consequence of h-prop elimination holds a weak form of the axiom of choice: the **principle of unique choice** UC. The principle UC essentially says that if $\forall(x : X)\, \exists!(y : Y)$ $P(x, y)$, where $\exists!$ means "exists and unique", then $\exists(f : X \to Y)\, \forall(x : X)\, P(x, f(x))$. h-prop elimination is a stronger form of UC, which, in particular, defines functions which compute at constructors. This implies, for example, that every bijection has an inverse with computational properties.

The **axiom of choice** AC obtained by dropping uniqueness in UC is not provable in Arend, but can be consistently assumed.

(g) h-prop elimination and the univalence axiom imply that Prop is isomorphic to totalities of h-propositions: $\mathsf{Prop} \cong \sum_{(A:\mathsf{Set}\,n)} \mathsf{isProp}(A)$ for all $n \in \mathbb{N}$. It follows from provability of $\mathsf{isProp}(A) \to \mathsf{nonempty}(A) = A$ using h-prop elimination and the univalence axiom. Moreover, with LEM the universe Prop is isomorphic to Bool.

To sum up, the logic of Arend provides a basis for either constructive or classical univalent formalization of mathematics where:

(a) Univalence axiom and the structure identity principle hold.

(b) The principle of unique choice UC holds in the computational form of h-prop elimination for Prop.

(c) The axiom of choice AC and the law of excluded middle LEM are not accepted in arend-lib, but can be consistently assumed in some formalizations.

Universes $\mathsf{Set}\ n$ are characterized by the following properties:

- For $X : \mathsf{Set}\ n$ and $x, y : X$ the type $x = y$ lies in Prop. In particular, sets satisfy uniqueness of identity proofs $\mathsf{UIP}(X) := \prod_{(x,y\,:\,X)} \mathsf{PI}(x = y)$.
- $\mathsf{Set}\ n \subset \mathsf{Set}\ (n + 1)$, $\mathsf{Set}\ n : \mathsf{Set}\ (n + 1)$ and there is a hierarchy of universes $\mathsf{Type}\ n$ of all types satisfying $\mathsf{Set}\ n \subset \mathsf{Type}\ n$.
- As in the case of propositions, there is a semantic notion of an **h-set**, which is any type A for which $\mathsf{UIP}(A)$ is provable. And similarly to propositions, definitions with a proof of $\mathsf{isSet}(A) := \mathsf{UIP}(A)$ can be placed in Set, and the elimination for truncated inductive types in $\mathsf{Set}\ n$ is confined to h-sets (**h-set elimination**). This implies $\mathsf{Set}\ n \cong \sum_{(A:\mathsf{Type}\,n)} \mathsf{isSet}(A)$.

3. **Homotopy levels and polymorphism.** Sets in Arend are types A such that their equality type $x =_A y$ is a proposition. This construction can be iterated by considering 1-types such that their equality type is a set, 2-types such that their equality type is a 1-type and so on. This leads to the notion of a **homotopy level** of a type: we say that propositions are of homotopy level -1 and a type is of homotopy level $k \in \mathbb{N}$ if its equality type is of homotopy level $k - 1$. We say that a type is of homotopy level ∞ if no such k exists.

   In Arend universes $\mathsf{Type}\ (n, k)$ are also parameterized by the homotopy level k (apart from the usual predicative level n). As in the case of Prop and $\mathsf{Set}\ n$ there is a universe placement mechanism for definitions, k-type elimination and an equivalence between $\mathsf{Type}\ (n, k)$ and the type of all semantic k-types: $\mathsf{Type}\ (n, k) \cong \sum_{(A:\mathsf{Type}\ n)} \mathsf{isOfHLevel}(A, k)$.

   This provides a basis for allowing definitions to be polymorphic on the homotopy level.

4. **Subtyping and dependent record types**. For some pairs (A, B) of types in Arend holds $A < B$ (subsumptive subtyping), which means that the typing relation $a : A$ implies the typing relation $a : B$. Subtyping is involved in two mechanisms: in cumulativity of the hierarchy of universes and, more significantly, in the system of dependent record types.

   Cumulativity of the hierarchy of universes means that bigger universes contain types from smaller universes: $\mathsf{Type}\,(p, h) < \mathsf{Type}\,(p', h')$ if and only if $p \leq p'$ and $h \leq h'$. In particular $\mathsf{Prop} < \mathsf{Set}\,0 < \mathsf{Set}\,1 < \ldots$

   Dependent record types provide a basis for working with hierarchies of mathematical structures by means of bundling data and creating new structures as extensions or specifications of other structures. Record types $C\{x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n\}$ (where A_i may depend on $x_1, \ldots, x_{i-1}$) are similar to sigma-types: their elements are also dependent tuples $(a_1 : A_1, a_2 : A_2[a_1/x_1], \ldots, a_n : A_n[a_1/x_1, \ldots, a_{n-1}/x_{n-1}])$. Variables $x_1, \ldots, x_n$ are called the *fields* of C. The difference from sigma-types is that a new record type D can be formed as an extension of C in which case D becomes a subtype of C. Record type D can extend C with new fields $x_{n+1}, \ldots, x_{n+k}$ or with **manifest fields** of the form $x_i \Rightarrow a_i$, $i \in \{1, \ldots, n\}$, which assign a specific value for fields of C. If D contains $x_i \Rightarrow a_i$ then its objects would be tuples where i-th component is computationally equal to a_i.

   Some of the properties of record types:

   (a) Record type extensions model the concept of inheritance in programming languages. In Arend multiple inheritance is allowed: a record type can be an extension and therefore a subtype of several record types. Multiple inheritance together with manifest fields provide a flexible framework for grouping data associated to mathematical objects using sharing, extensions and subtyping.

   (b) Record extensions consisting of just manifest fields can be anonimous. This implies that records' fields also behave like parameters: if $C\{x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n\}$ is a record, the application $C\,a_1\,\ldots\,a_k$ is the anonimous extension $C\{x_1 \Rightarrow a_1, \ldots, x_k \Rightarrow a_k\}$.

   (c) If D is an extension of C and $x : A$ is a field of C then the type of the field x can be overriden in D and changed to a subtype B: $x : B$, $B < A$.

   For example, a record SetHom for morphisms in the category of sets would contain fields Dom, Codom for domain and codomain which are sets. A record $\mathsf{RingHom}$ of morphisms in the category of rings can be defined as an extension of SetHom with types of Dom, Codom changed to the type of rings.

   (d) Although propositions in Arend are not computationally proof irrelevant in general, a form of computational proof irrelevance holds for propositions inside record types. Namely, some fields whose type is a proposition can be declared **properties** in which case their value will be irrelevant in comparing elements of records.

   For example, a record Cat for categories would have data fields for objects, morphisms, identity morphism and composition and property fields for associativity of composition and neutrality of identity morphism. One consequence of this would be that if $C : \mathsf{Cat}$ then C is computationally equal to $(C^{op})^{op}$.

5. **Arrays.** Arend has built-in type Array of arrays which subsumes the standard inductive type $\mathsf{List}\,(A : \mathsf{Type})$ of lists of varying length, the standard indexed inductive type $\mathsf{Vec}\,(A : \mathsf{Type})\,(n : \mathsf{Nat})$ of lists of fixed length n and the type of functions $\mathsf{Fin}\,\mathsf{len} \to A$, where $\mathsf{Fin}\,(\mathsf{suc}\,n) : \mathsf{Set}$ is the type corresponding to the finite set $\{0, \ldots, n\}$. The type Array is a special kind of record with the fields: $A : \mathsf{Type}$ for elements, $\mathsf{len} : \mathsf{Nat}$ for length and an indexing function $\mathsf{at} : \mathsf{Fin}\,\mathsf{len} \to A$. Types List and Vec can be defined via Array as anonymous extensions: $\mathsf{List}\,(A : \mathsf{Type}) := \mathsf{Array}\,A$, $\mathsf{Vec}\,(A : \mathsf{Type})\,(n : \mathsf{Nat}) := \mathsf{Array}\,A\,n$. The type $\mathsf{Array}\,A\,n$ is obviously isomorphic to the type of functions $\mathsf{Fin}\,n \to A$, but it satisfies additional computational equalities which is the central motivation for introducing $\mathsf{Array}\,A\,n$:

$$\frac{\begin{array}{lll}\Gamma \vdash A & \Gamma \vdash f : \mathsf{Fin}\,(n+2) \to A & \Gamma \vdash f\,n \equiv g\,n \\ \Gamma \vdash n : \mathsf{Nat} & \Gamma \vdash g : \mathsf{Fin}\,(n+2) \to A & \Gamma, x : \mathsf{Fin}\,(n+1) \vdash f\,x \equiv g\,x\end{array}}{\Gamma \vdash f \equiv g}$$

There is also dependent version DArray with $A : \mathsf{Fin\ len} \to \mathsf{Type}$ and $\mathsf{at} : \prod_{(i:\mathsf{Fin\ len})} A\, i$.

We now explain what these modifications bring to MLTT and how it affects formalisation in general, not necessarily in the context of homotopy theory. We then outline some nice properties of the type theory of Arend specific for formalisations in homotopy theory.

2.3 Properties of the theory of Arend relevant for set-theoretic formalization

2.3.1 Function extensionality

Function extensionality principle says that if two functions are pointwise equal then they are equal:

$$\frac{\begin{array}{ccc} \Gamma \vdash A & \Gamma \vdash f : \prod\limits_{x:A} B\, x & \\ \Gamma \vdash B & \Gamma \vdash g : \prod\limits_{x:A} B\, x & \Gamma, x : A \vdash p : f\, x = g\, x \end{array}}{\Gamma \vdash \mathsf{FunExt}(f, g, \lambda x.p) : f = g}$$

This principle cannot be proven in MLTT and has to be added as an axiom. It is known to be provable from univalence axiom in HoTT, but the proof is not completely straightforward ([28], Section 4.9). Definition of equality type in HoTT-I allows to define FunExt very easily:

$$\mathsf{FunExt}(f, g, \lambda x.p) :\equiv \mathsf{path}(\lambda i.\lambda x.p@i)$$

Moreover, this definition of FunExt is well behaved computationally. As a map between pointwise equalities to equalities between functions FunExt admits an obvious inverse $\lambda p.\lambda x.\mathsf{path}(\lambda i.p@i\, x)$ such that composition on both sides is computationally the identity.

Derivation of FunExt in cubical type theories is also elementary (for example, in Cubical Agda [29]). In Lean FunExt is derived using built-in axioms for quotient types and propositional extensionality.

2.3.2 Quotient types

Given a type A and a relation $R : A \to A \to \mathsf{Prop}$[4] the **quotient type** A/R is the type obtained from A by identifying all $a, b : A$ such that $R(a, b)$ holds. More formally A/R can be characterised as a type satisfying the following properties:

1. There is a function $c : A \to A/R$.
2. (Universal property) For any type B any function $f : A \to B$ respecting R (that is such that $f(a) = f(b)$ whenever $R(a, b)$) factorizes uniquely through c: there exists a unique $h : A/R \to B$ such that $f = h \circ c$. The map h corresponds to an eliminator for A/R.

In terms of category theory A/R is the coequalizer of two projections $\sum_{(a,b\,:\,A)} R(a, b) \rightrightarrows A$. The concept of quotient type is well known to be hard to deal with in MLTT due to intensionality of equality [18, 12, 1]. Although quotient types can be defined in MLTT in some cases (say, for rationals) in general they are not definable in MLTT (say, for Cauchy reals)[18]. We briefly outline some of the proposed ways to introduce arbitrary quotient types:

1. **Setoids.** Instead of working with types one can work with types coupled with an equivalence relation [12, 1, 2]. If we think of types as sets, this corresponds to replacing sets with setoids. Quotient type would correspond to quotient setoid, which is defined simply by taking union of two equivalence relations. Coq, for example, has a well developed setoid infrastructure that helps to adopt this approach. However, this approach is rather impractical because of all the complications of working with a custom term rewriting system for equivalence relations (as opposed to built-in one for the equality type).

[4]Here Prop stands for the totality of propositions, which can be the universe of propositions like in Arend, Coq, Lean or just sort Type in pure MLTT

2. **Axioms for existence of quotient types.** Quotient types can be introduced purely axiomatically by simply assuming the statement phrasing that there exists a type A/R satisfying the universal property. The disadvantage of this approach compared to the previous one is that it is not constructive since assuming any kind of axioms destroys certain meta properties of MLTT such as canonicity. Moreover there would be no computational rules for A/R (unless built-in in type theory) which is inconvenient in practice. Nevertheless this approach is often used in Coq, Agda and Lean. In Lean the computational rules for A/R are built-in.

3. **Quotients of h-sets in the logic of h-propositions in HoTT.** Quotients can be defined for h-sets using subuniverse hProp of h-propositions and univalence (or extensionality for functions and h-propositions): if R is hProp-valued equivalence relation on an h-set $A : \mathsf{hSet}$, then A/R can be defined as the type of all $P : A \to \mathsf{hProp}$ that define an equivalence class of R in A. This definition rises the universe level which can be dealt with by means of *resizing rules* ([28], Section 6.10).

4. **Higher inductive types.** An appealing perspective on dealing with constructs like quotient types systematically was brought by HoTT in the form of a mechanism to introduce equalities in definitions of inductive types. Definitions of higher inductive types (HITs) in HoTT apart from ordinary constructors contain constructors for equalities called path constructors, and the modified elimination principle ensures the universal property (see Section 2.4 below and Chapter 6 in [28]).

   First, this allows to define the operation $\|A\|_0$ of *set-truncation* for types $A : \mathsf{Type}$ which produces h-set by trivializing the structure of the identity type on A and enforces UIP on $\|A\|_0$. The h-set $\|A\|_0$ can be defined as the higher inductive type with ordinary constructor $\mathsf{in}_{\mathsf{Set}}\,(x : A) : \|A\|_0$ and path constructor $\mathsf{trunc}\,(x, y : \|A\|_0)\,(p, q : x = y) : p = q$.

   Second, for an h-set A and hProp-valued relation R the quotient h-set A/R can be defined as $A/R := \|Q(A, R)\|_0$ where $Q(A, R)$ is the higher inductive type with ordinary constructor $\mathsf{in}_Q\,(x : A) : Q(A, R)$ and path constructor $\mathsf{equiv}\,(x, y : A)\,(p : R\,x\,y)\ :\ \mathsf{in}_Q\,x = \mathsf{in}_Q\,y$ [5]. The presence of computational rules analogous to those of ordinary inductive types makes it computationally better behaved solution than those based on axioms. On the negative side: some of the constructors in the inductive definition of A/R are required to be elements of a different type (the equality type) and quotients A/R are still not fully constructive. These issues have been solved in cubical type theories, but at the cost of making the theory two-level and significantly more complicated.

The type theory of Arend allows for particularly convenient definitions following the last two approaches.

Definition (3) can be stated in Arend with the universe Prop in place of hProp and Set in place of hSet. Since Prop in Arend is impredicative, resizing rules are unnecessary in this case. Meanwhile due to absence of elimination outside of Prop for propositions in Coq, Lean and Agda, definition (3) requires h-prop elimination for Prop there to be stated in computationless axiomatic form (which results in computationless quotient type) or involves subuniverses hProp and resizing rules.

Arend also improves on the approach (4). It turns HITs into ordinary inductive types with conditions. Instead of the path constructor $\mathsf{equiv}\,(x, y\ :\ A)\,(p\ :\ R\,x\,y)\ :\ \mathsf{in}\,x\ =\ \mathsf{in}\,y$ for $A/R\ :\ \mathsf{Set}$ we can define an ordinary constructor $\mathsf{equiv}\,(x, y\ :\ A)\,(p\ :\ R\,x\,y)\,(i\ :\ \mathbf{I})\ :\ A/R$ with conditions $\mathsf{equiv}\,x\,y\,p\,\mathsf{left} \equiv \mathsf{in}\,x$ and $\mathsf{equiv}\,x\,y\,p\,\mathsf{right} \equiv \mathsf{in}\,y$. Moreover instead of using set-truncation HIT Arend allows to simply declare A/R to be a truncated inductive type in Set with elimination restricted to h-sets. Unfortunately Arend's implementation of the approach (4) to the definition of a quotient type has the same disadvantages as the axiomatic approach (2) as is implemented in Lean. However it is more general and allows sometimes to simplify things by defining quotients not via A/R but directly as a inductive type with conditions as, for example, in the case of type of integers (see the beginning of Section 2.2) or the type of univariate polynomials in arend-lib (see Section 4.1.3).

[5]The untruncated type $Q(A, R)$ is not an h-set in general

2.3.3 Isomorphic types are equal

One important property of MLTT is that it cannot distinguish between isomorphic types [6]. Namely, if A and B are isomorphic types and $P\,X$ is a property of types then $P\,A$ and the negation of $P\,B$ cannot be simultaneously proven in MLTT. This can be shown by constructing a homotopical model of MLTT where existence of isomorphism between A and B implies existence of elements in $A = B$ [14].

In fact, a stronger statement holds for MLTT: all properties of types are stable under isomorphism. Namely, for any predicate $P\,X$ on types which is defined also for X outside of universes[7] there is a function $[\![P]\!](A, B : \mathsf{Type}) : (A \cong B) \to (P\,A \cong P\,B)$. This is called *external univalence* - a metatheoretic property of theories capturing preservation of isomorphisms. It is metatheoretic the construction of $[\![P]\!]$ is dependent on the structure of P and is not uniform. External univalence can be shown using homotopical inverse diagram construction [15] or univalent parametricity translation [27]. It has been proven in the general setting of second-order generalized algebraic theories in [4] which implies that it holds for any type theory with identity types and any standard choice of type formers like $\sum$-, $\prod$-types, universes, inductive types and so on.

It is thus natural to avoid the routine of proving stability for each particular $P\,X$ and pass to an extension or a modification of MLTT where univalence holds *internally* or more precisely where $A \cong B \to A = B$ holds.

The simplest minimal such extension of MLTT would be just MLTT with $\mathsf{ua} : A \cong B \to A = B$ added as an axiom. This is essentially the way univalence is introduced to HoTT-I: an isomorphism between types can be converted to equality between types by built-in function iso. A small advantage of HoTT-I over MLTT with the axiom is the presence of built-in computational rules for iso that make the use of univalence a bit more convenient.

Example: types of unary and binary natural numbers. A simple classical example of isomorphic types, where a mechanism for transferring proofs of properties from one to the other would be desirable, is given by different representations of natural numbers: unary and binary. This example is rather relevant for the formal verification: binary numbers are computationally efficient and are used in formal descriptions of algorithms whereas unary numbers are easier to reason about.

Unary natural numbers are defined as the standard inductive type Nat with constructors $0 : \mathsf{Nat}$ and $\mathsf{suc} : \mathsf{Nat} \to \mathsf{Nat}$. This definition is convenient for proving properties of natural numbers via standard induction but is computationally inefficient.

The type $\mathsf{Nat}_{\mathsf{Bin}}$ of natural numbers represented as binary strings without trailing zeros can be defined as an inductive type with constructors $0_{\mathsf{Bin}} : \mathsf{Nat}_{\mathsf{Bin}}$ and $\mathsf{pos} : \mathsf{Bin}_{+} \to \mathsf{Nat}_{\mathsf{Bin}}$ were Bin_{+} is an inductive type with constructors $1_{\mathsf{Bin}} : \mathsf{Bin}_{+}$ and $\mathsf{digit} : \mathsf{Fin}\,2 \to \mathsf{Bin}_{+} \to \mathsf{Bin}_{+}$. For example, the binary number 1010 is represented as the term $\mathsf{pos}\,(\mathsf{digit}\,0\,(\mathsf{digit}\,1(\mathsf{digit}\,0\,1_{\mathsf{Bin}}))) : \mathsf{Nat}_{\mathsf{Bin}}$. In contrast to the unary definition Nat above, this definition of natural numbers is computationally efficient, but proving even basic properties of $+_{\mathsf{Bin}}$ and $*_{\mathsf{Bin}}$ defined in computationally efficient way becomes rather involved.

Problems such as enabling Nat to be used for proving properties of $\mathsf{Nat}_{\mathsf{Bin}}$ have been central to a long line of research on mechanisms for transferring proofs of theorems between isomorphic types [21, 5, 32]. At the heart of many approaches lies the concept of parametricity [23] which is closely tied to developments on free theorems about observational equivalences [30], data refinements for free [5] and proofs for free [3].

Recent work [27] demonstrates how certain limitations of parametricity-based approaches can be addressed by combining parametricity and univalence. Parametricity enables a *white-box* translation of definitions and proofs of theorems between equivalent types by leveraging the internal structure of the terms being translated and their types. In contrast, univalence provides a universally applicable *black-box* translation through transport along equalities, such as $\mathsf{ua}(f) : A = B$ for $f : A \cong B$. Each approach, however, has its limitations when used in isolation.

For instance, univalent transport of an efficient function $\mathsf{Nat}_{\mathsf{Bin}} \to \mathsf{Nat}_{\mathsf{Bin}}$ results in a function $\mathsf{Nat} \to \mathsf{Nat}$ that retains its efficiency. However, the convenient induction principle for Nat becomes inapplicable to it. On the other hand, parametric transport can utilize correspondences between different implementations of functions, such as $+$ and $+_{\mathsf{Bin}}$. Nevertheless, the applicability of this approach in dependently-typed settings

[6]We prefer to call "isomorphism" here what is usually called "equivalence" in HoTT since we prefer to turn to pre-HoTT terminology while discussing applications to set-theoretic formalization

[7]In particular, $P\,X$ can be arbitrary closed predicate

is constrained due to the critical role of type-level computations. In particular, a dependent type $A(n)$ defined using eliminator for Nat satisfies some computational rules which would be lost for its analogue for $\mathsf{Nat_{Bin}}$ and that would create problems when translating functions $\prod_{n:\mathsf{Nat}} A(n)$ defined using eliminator for Nat.

As suggested in [27], these problems can be addressed by devising combined approach of univalent parametricity, which intertwines white-box and black-box translations in a complementary and effective manner.

Isomorphic structures. MLTT likewise cannot distinguish between various isomorphic structures on top of types such as groups, rings, topological spaces and so on. In other words MLTT cannot distinguish between isomorphic objects in various categories admitting a natural forgetful functor to the category of sets. And typically the function iso can be used to derive equality of isomorphic objects and freely transfer properties of the one to properties of the other. This is known as the **structure identity principle** in Univalent Foundations ([28], Section 9.8).

Example: definition of schemes. We now give a concrete example from Arend standard library arend-lib, where the function iso helps to avoid proving stability of a property under isomorphisms. The example is concerned with the definition of a **scheme** from the part of arend-lib devoted to algebraic geometry. In this particular example the need to prove stability of the property of "being locally ringed space" under isomorphisms is caused by the impossibility to define in MLTT the general factor ring R/I construction in such a way that $R/\emptyset = R$ and not just $R/\emptyset \cong R$.

Let us first briefly recall some definitions involved in the definition of a scheme:

- **Locale ($\approx$ Topological space).** Formalization of algebraic geometry in arend-lib does not deal with topological spaces and replaces them with closely related **locales**, certain kind of posets that can be thought of as posets of open sets.
- **Ringed locale** $(L, \mathcal{O}_L)$ is a locale L together with a sheaf of rings $\mathcal{O}_L$ on L, that is with a functor (presheaf) $\mathcal{O}_L : L^{\mathrm{op}} \to \mathsf{Ring}$ satisfying certain properties. The sheaf $\mathcal{O}_L$ is called **structure sheaf** and can be thought of as a functor mapping an open set to the ring of regular functions on this open set.
- **Locally ringed locale** $(L, \mathcal{O}_L)$ is a ringed locale such that, if we think of locale as a topological space, for every point $x \in L$ the colimit $\mathsf{stalk}_x(\mathcal{O}_L) := \mathsf{colim}_{x \in U} \mathcal{O}_L(U)$ over all open sets U of L containing x is a local ring.
- **Spectrum of a ring.** For a commutative ring R its spectrum $\mathsf{Spec}\, R$ is a topological space with prime ideals of R as points and arbitrary ideals as closed sets. It is a locally ringed locale with structure sheaf given by localisation $\mathcal{O}_R(R\backslash I) := I^{-1}R$.

We have now everything that is needed for the definition of a scheme. An **affine scheme** $(X, \mathcal{O}_X)$ is a ringed locale of the form $(Spec\, R, \mathcal{O}_R)$ for some commutative ring R. A **scheme** is a ringed locale $(X, \mathcal{O}_X)$ which can be covered by affine schemes, that is there exists a system of open sets $\{U_i\}$ such that $X = \cup U_i$ and for each i the ringed locale $(U_i, \mathcal{O}_X \restriction_{U_i})$ is affine scheme.

It can be proven that all schemes are locally ringed locales: for affine schemes it follows from the fact that spectra are locally ringed and for arbitrary schemes from the fact that a locale covered by locally ringed locales is locally ringed. But the translation of the property of being locally ringed from spectra to other schemes depend on whether we require $(X, \mathcal{O}_X) = (\mathsf{Spec}\, R, \mathcal{O}_R)$ or $(X, \mathcal{O}_X) \cong (\mathsf{Spec}\, R, \mathcal{O}_R)$ in the definition of affine schemes and whether $(U_i, \mathcal{O}_X \restriction_{U_i}) = (Y_i, \mathcal{O}_{Y_i})$ or $(U_i, \mathcal{O}_X \restriction_{U_i}) \cong (Y_i, \mathcal{O}_{Y_i})$ for affine $(Y_i, \mathcal{O}_{Y_i})$ in the definition of schemes. With iso we can assume equalities everywhere and avoid any extra complications whatsoever. Without iso one has to choose between the following scenarios each of which bring some complications to formalisation:

1. If isomorphism is used in any of the places above, then one has to prove that the predicate "to be locally ringed" is stable under isomorphism.

2. If in both places are equalities then affine schemes are not schemes since $(X, \mathcal{O}_X \restriction_X)$ is only isomorphic to $(X, \mathcal{O}_X)$ and not equal . Therefore properties proved for schemes would not automatically be proven for affine schemes.

3. One can modify definition of a scheme and say that either $(X, \mathcal{O}_X)$ is affine scheme or can be covered by affine schemes. But that means that case distinction would be necessary every time when proving some property of schemes.

There are no obstacles in the second scenario above in the presence of iso, of course, since it can be used to generate equality $(X, \mathcal{O}_X \restriction_X) = (X, \mathcal{O}_X)$.

See Section 4.1.3 for more details on how it is manifested in arend-lib.

2.3.4 Logic and set theory

Since Prop is impredicative and due to univalence, the category of sets in Arend formed by universes $\mathsf{Set}\,n$ is an elementary topos. If Prop is extended with LEM and AC then the category of sets becomes a model of Lawvere's Elementary Theory of the Category of Sets ([28], Section 10.1). It is possible to interpret the constructive set theory CZF in $\mathsf{Set}\,n$ and ZFC in case Prop is extended with LEM and AC ([28], Section 10.1, [9]).

2.3.5 Hierarchies of structures

Formalization of mathematics crucially relies on mechanisms for grouping data and properties of mathematical structures. In languages based on variants of MLTT this is often done by means of dependent record types supplemented with typeclasses and instance inference mechanism. Dependent record types and typeclasses in such languages are closely related to module systems and typeclasses in functional programming languages [17, 11, 10]. Here we focus on record types since the mechanism of typeclasses does not require any extensions of the core type theory, we discuss it in more detail in Section 3.2.

Record types can be used, for example, to define the type of monoids as follows:

$$\begin{aligned}\mathsf{Monoid}\,\{&E : \mathsf{Set},\ \mathsf{ide} : E,\ * : E \to E \to E,\\ &\mathsf{ide\text{-}left}\,(x : E) : \mathsf{ide} * x = x,\ \mathsf{ide\text{-}right}\,(x : E) : x * \mathsf{ide} = x,\\ &*\text{-}\mathsf{assoc}\,(x\,y\,z : E) : (x * y) * z = x * (y * z)\}\end{aligned}$$

Elements X : Monoid are essentially tuples containing as data the carrier set $X.E$, monoid identity $X.\mathsf{ide}$ and binary operation $X.*$ together with proofs that this data defines a monoid structure on $X.E$.

The challenges that arise when designing a dependently typed language with record types include the support of multiple inheritance, subtyping, partial implementations and sharing. For example, a field is both a commutative local ring and a GCD domain so it is natural to define the type Field of fields as an extension of LocalCRing and GCDDomain which requires multiple inheritance. Apart from making Field inherit the data and properties of LocalCRing and GCDDomain it involves making Field a subtype of the two. Finally, in the constructive setting definition of a domain contains apartness predicate $x\#0$ as part of data[8] which in Field can be defined as invertibility[9] $x\#0 \Rightarrow \mathsf{isInvertible}(x)$ and this requires partial implementations. The tightness of the apartness relation then exactly says that non-invertible elements equal to 0 and is left as unimplemented field. We discuss this example in more detail in Section 3.2.

In Arend these challenges are addressed both at the level of the core type theory and at a higher level. As described in Section 2 the type theory of Arend has extensions for subsumptive subtyping $A < B$ and manifest fields $x \Rightarrow a$ for record types where the latter provides a mechanism for partial implementations and sharing. From theoretical point of view such extensions of MLTT were studied, for example, in [6]. There are also implementations: manifest fields are supported, for example, by the proof assistant Matita

[8]With LEM $x\#0$ must be the same as $x \neq 0$

[9]A constructive (Heyting) field is by definition a commutative local ring such that non-invertible elements equal to 0

[25]. These extensions are extremely useful for working with hierarchies of so-called *bundled* or *semi-bundled* definitions where all or almost all operational data of a mathematical structure and its defining predicates are placed in fields of a record (and the rest of data is in parameters). For example, the definition of Monoid above is fully bundled, its less bundled variants would be records with parameters $\mathsf{Monoid}'\,(E : \mathsf{Set})$, $\mathsf{Monoid}''\,(E : \mathsf{Set})\,(ide : E)$, $\mathsf{Is\_Monoid}\,(E : \mathsf{Set})\,(ide : E)\,(op : E \to E \to E)$. Definitions like $\mathsf{Is\_Monoid}$ are typically classified as *unbundled* since all its data components are parameters. Unbundled definitions allow to reduce partial implementations to usual applications, though at the price of losing key benefits of record types such as being a type of tuples with named projections. Still with the mechanism of typeclasses and instance inference unbundled approach can be made viable as suggested, for example, in [26].

Without manifest fields possibilities for defining record extensions become much more limited even in unbundled case. For example, manifest fields allow to define record Ring as an extension of $\mathsf{Semiring}$. Some of defining properties of $\mathsf{Semiring}$, namely $x * 0 = 0$ and $0 * x = 0$, become derivable in Ring and therefore these fields should be implemented (see also Section 3.2). Without manifest fields, in bundled or unbundled case, typically one has to define more of the small, atomic records and combine them to define various composite records, the option of constructing composite records as specializations of other composite records becomes unavailable.

Arend language has numerous high level constructs on top of record types which we mainly discuss in Section 3.2: typeclasses, coercive subtyping, anonymous extensions and so on. Parameters of records is also a high level construct in Arend: the mechanism of manifest fields allows to translate parameters to fields in the core type theory.

The development of major theorem provers from MLTT family such as Lean, Coq and Agda has been avoiding subsumptive subtyping and manifest fields in the core theory thus making record types the same as sigma types. Subsumptive subtyping there is replaced with coercive subtyping $A <_c B$ in which case typechecker inserts an application $c\,a : B$ of some function $c : A \to B$ whenever a term $a : A$ is used in a context where a term of type B is expected. There are also ways to deal with manifest fields without extending the core type theory as studied, for example, in [20].

2.3.6 Arrays

The primary application of the type Array is that it significantly simplifies recursion and induction for nested inductive types involving inductive types $\mathsf{List}\,A$ and $\mathsf{Vec}\,A\,n$. To illustrate this consider inductive type of terms $\mathsf{Term}\,(F : \mathsf{Set})\,(a : F \to \mathsf{Nat})$ with the constructor $\mathsf{fun}\,(f : F)\,(v : \mathsf{Vec}\,(\mathsf{Term}\,F\,a)\,(a\,f))$ – constructor for the function application term $f\,v_1\,\ldots\,v_{a(f)}$, where f is a functional symbol of arity $a(f)$ and $v_1, \ldots, v_{a(f)}$ are terms.

In this case the nesting occurs when the type Term, which is being defined, is used as a parameter of type Vec in its constructor func. By $\mathsf{Vec}\,(A : \mathsf{Type})\,(n : \mathsf{Nat})$ here we mean the standard type of lists of length n represented as indexed inductive type generated by the constructors $\mathsf{nil} : \mathsf{Vec}\,A\,0$ and $\mathsf{cons}\,(a : A)\,(v : \mathsf{Vec}\,A\,n) : \mathsf{Vec}\,A\,(\mathsf{suc}\,n)$.

The problem here is that it is unclear what should be the eliminator $\mathcal{E}_{\mathsf{Term}}$ for Term. Some obvious choices would not enable proofs and constructions by induction on subterms since in the induction step there is no way to refer to i-th element of v[10]. The type Array allows to work with v as if it were a function and simply form applications $v\,i$. But Array is not just the type of functions: its computational behavior is analogous to the one for Vec (see Section 2.2). For example, if $x_1 \equiv x_2$ and $y_1 \equiv y_2$ then $f\,x_1\,y_1 \equiv f\,x_2\,y_2$.

2.4 Type theory of Arend under types as homotopy types viewpoint

1. **Higher inductive types.** Higher inductive types (HITs) provide a mechanism for inductive definitions of types with non-trivial homotopical structure on them. HITs are akin to CW complexes which are defined by specifying cells of various dimensions and the way they glued. The original approach to defining HITs in HoTT was as follows: a HIT D contains ordinary constructors, which are generating

[10]At induction step one would like to use values $G(v_i)$ to construct the value $G(f\,v_1 \ldots v_{a(f)})$ while defining G

points of D, and path constructors, which are generating paths of D and have types of the form $l =_D r$, $s =_{(l=_D r)} t$ and so on ([28], Chapter 6).

For example, the circle $\mathbb{S}^1$ can be defined as HIT with the following constructors:

(a) Point constructor $\mathsf{base} : \mathbb{S}^1$ for the base point.
(b) Path constructor $\mathsf{loop} : \mathsf{base} =_{\mathbb{S}^1} \mathsf{base}$ for the loop.

A definition of the torus $\mathbb{T}^2$ would involve cells of dimension 2, for example:

(a) Point constructor $\mathsf{point} : \mathbb{T}^2$.
(b) Path constructors $\mathsf{loop}_1, \mathsf{loop}_2 : \mathsf{point} =_{\mathbb{T}^2} \mathsf{point}$ for two loops.
(c) Path constructor $\mathsf{face} : \mathsf{loop}_1 \cdot \mathsf{loop}_2 =_{(\mathsf{point} =_{\mathbb{T}^2} \mathsf{point})} \mathsf{loop}_2 \cdot \mathsf{loop}_1$ defining a cell of dimension 2, where $\mathsf{loop}_1 \cdot \mathsf{loop}_2$ is concatenation of paths.

The recursion principle for HITs is a straightforward generalisation of the one for ordinary inductive types. For example, in order to define a function from $\mathbb{S}^1$ to a type A one needs to specify a point $a : A$ and a loop $l : a =_A a$ in A and the recursor gives a function $\mathsf{rec}^{a,l}_{\mathbb{S}^1} : \mathbb{S}^1 \to A$ such that $\mathsf{rec}^{a,l}_{\mathbb{S}^1}(\mathsf{base}) \equiv a$ and $\mathsf{pmap}\ \mathsf{rec}^{a,l}_{\mathbb{S}^1}\ \mathsf{loop} = l$, where pmap is an obvious function mapping a path between arguments to a path between images of a function . Dependent case, that is the induction principle, is also not hard to state, but it is a bit more subtle ([28], Section 6.2).

With this approach to HITs it is far from obvious how to formulate a rigorous schema of general inductive definitions of this kind.

The interval type in Arend allows to use ordinary **constructors with conditions** instead of path constructors. For example, $\mathbb{T}^2$ can be defined in Arend as an **inductive type with conditions** with the following constructors:

(a) $\mathsf{point} : \mathbb{T}^2$.
(b) $\mathsf{line}_1\ (i : \mathbf{I}) : \mathbb{T}^2$, $\mathsf{line}_2\ (i : \mathbf{I}) : \mathbb{T}^2$ with conditions $\mathsf{line}_{1(2)}\ \mathsf{left} \equiv \mathsf{point}$ and $\mathsf{line}_{1(2)}\ \mathsf{right} \equiv \mathsf{point}$.
(c) $\mathsf{face}\ (i : \mathbf{I})\ (j : \mathbf{I}) : \mathbb{T}^2$ with conditions:
  - $\mathsf{face}\ \mathsf{left}\ j \equiv \mathsf{line}_2\ j$.
  - $\mathsf{face}\ \mathsf{right}\ j \equiv \mathsf{line}_2\ j$.
  - $\mathsf{face}\ i\ \mathsf{left} \equiv \mathsf{line}_1\ i$.
  - $\mathsf{face}\ i\ \mathsf{right} \equiv \mathsf{line}_1\ i$.

The recursion and induction principles are just the standard ones with the requirement that conditions must be preserved by the function being defined. The general schema of inductive definitions of this kind can be easily formulated: it is the standard schema plus specification of valid forms of conditions, which is rather simple. Definitions of HITs in Arend are pretty similar to definitions of HITs in cubical type theories, however there are some differences ([31], Section 2.4.4).

Elimination principles for HITs, as formulated in [28], can be derived in Arend, so all HoTT book style proofs can be carried out in Arend. But often inductive types with conditions allow for much simpler proofs. To illustrate this consider a proof of $\mathbb{T}^2 \cong \mathbb{S}^1 \times \mathbb{S}^1$ in Arend. The definition of mutually inverse functions $f : \mathbb{T}^2 \to \mathbb{S}^1 \times \mathbb{S}^1$ and $g : \mathbb{S}^1 \times \mathbb{S}^1 \to \mathbb{T}^2$ is completely straightforward:

(a) Using elimination for $\mathbb{T}^2$ the function f is defined by its values on constructors: $f(\mathsf{point}) :\equiv (\mathsf{base}, \mathsf{base})$, $f(\mathsf{line}_1\, i) :\equiv (\mathsf{loop}\, i, \mathsf{base})$, $f(\mathsf{line}_2\, j) :\equiv (\mathsf{base}, \mathsf{loop}\, j)$, $f(\mathsf{face}\, i\, j) :\equiv (\mathsf{loop}\, i, \mathsf{loop}\, j)$.
(b) Using double elimination for $\mathbb{S}^1$ the inverse g is defined by its values on constructors: $g(\mathsf{base}, \mathsf{base}) :\equiv \mathsf{point}$, $g(\mathsf{loop}\, i, \mathsf{base}) :\equiv \mathsf{line}_1\, i$, $g(\mathsf{base}, \mathsf{loop}\, j) :\equiv \mathsf{line}_2\, j$, $g(\mathsf{loop}\, i, \mathsf{loop}\, j) :\equiv \mathsf{face}\, i\, j$.

The identities $g(f\, x) = x$ and $f(g\, y) = y$ are proven by reflexivity for each of the constructors and thus can be proven trivially using elimination. In contrast, the HoTT book style proof of $\mathbb{T}^2 \cong \mathbb{S}^1 \times \mathbb{S}^1$ is rather involved ([28], Exercise 6.3).

3 Arend language

In this section we present the most essential language constructs of Arend.

3.1 Levels and universe polymorphism

Since there is no way to assume consistently the existence of the set Set of all sets any construction that aims at producing a set out of the totality of all sets has to be applied to a universe of small sets Set 0 producing a set in a universe of large sets Set 1. Then such constructions can typically be iterated and produce a set in the universe Set $i+1$ of level $i+1$ out of the universe Set i of level i. And typically this routine is straightforward and can be automated.

Arend language allows to use Set and Type in practice as if they were types of types and delegate the routine of dealing with levels to the typechecker. Typechecker replaces Set and Type with hierarchies of universes, converts definitions accordingly and checks if definitions are not contradictory because of a circularity in universe hierarchy. This is achieved by the following mechanisms:

1. **Universe polymorphism.** Definitions are made polymorphic on universe levels: there are implicit level parameters associated with each definition.

2. **Level inference.** In each occurrence of Set or Type typechecker infers level according to some natural procedure.

For example, the following code typechecks in Arend:

```
\func id {A : \Set} (a : A) => a
\func id-test-set : \Set => id \Set
\func id-test-id : \Set => id (id \Set)
```

Arend typechecker unfolds definitions of `id`, `id-test-set` and `id-test-id` as functions each with its own implicit level parameter `\lp`. The unfolded definition of `id` takes as a parameter a type `A : \Set \lp` in a universe of level `\lp`. In the body of the function `id-test-set` the function `id` is applied to the universe `\Set \lp`, the type of `id-test-set` is thus `\Set (\suc \lp)`. In the body of `id-test-id` inner `id` is of type `\Set (\suc \lp) -> \Set (\suc \lp)`, the outer `id` thus belongs to the universe of level `\suc \suc \lp`.

Levels of universes can also be specified explicitly in Arend using **level expressions** involving level variables, `0` and operations `\suc l` and `\max l l'`. For example, the code above can equivalently be written as follows:

```
\func id {A : \Set \lp} (a : A) => a
\func id-test-set : \Set (\suc \lp) => id (\suc \lp) (\Set \lp)
\func id-test-id : \Set (\suc \suc \lp) => id (\suc \suc \lp) (id (\suc \lp) (\Set \lp))
```

Polymorphic definitions can have several predicative level parameters in which case they must be linearly ordered. For example, it is quite useful to have separate and comparable level parameters for objects and morphisms in the definition of a category:

```
\class Cat \plevels obj >= hom
```

Universe management in Arend is rather similar to the one in Lean and late versions of Coq. It is quite different in Agda: Agda has considerably less automation, one has to be much more explicit about levels. Also, unlike level expressions in Agda where they are terms of type `Level`, level expressions in Arend are not typed.

3.1.1 Homotopy levels

The universe `\Prop` and the hierarchy `\Set n` are contained in the double hierarchy `\Type n k` parameterized additionally by the homotopy level `k`: `\Prop` is `\Type n -1` and `\Set n` is `\Type n 0`. Definitions in Arend can be made polymorphic on the homotopy level as well: each definition by default has implicit homotopy level parameter `\lh` apart from `\lp`. Most of the constructs for predicative levels described above are likewise applicable to homotopy levels: for example, the language of level expressions is the same for both predicative and homotopy levels.

3.1.2 Definitions in Prop, Set n and Type (n, k)

By default a definition is inferred to be of some homotopy level according to a certain rules and is placed in the corresponding universe. However one often needs to use a definition which is not inferred by the typechecker to be in `\Prop`, `\Set` or `\Type n k` to define a proposition, a set or a k-type in the corresponding universe. There are two ways to do this in Arend:

1. **Use level**. Assume `D` is a definition of a record, class or inductive type is provably a k-type which means that the predicate `isOfHLevel D k` is provable. In that case `D` can be turned into a definition in `\Type n k` by means of extending it with a proof of `isOfHLevel D k` which is done in Arend via `\use \level` construct. In particular, extending `D` with a proof of `isProp D` or `isSet D` would place `D` in `\Prop` or `\Set` respectively. For example, the predicate `nonempty A` can be defined in `\Prop` as follows:

   ```
   \data nonempty (A : \Type) : \Prop
     | in A
     | trunc (a a' : nonempty A) : a = a' -- just another syntax for the constructor
                                          -- trunc (a a' : nonempty A) (i : I) with
                                          -- conditions trunc a a' left => a
                                          --             trunc a a' right => a'
     \where {
       \use \level levelProp {A : \Type} (a a' : nonempty A) : a = a' => path (nonempty a a')
     }
   ```

   As mentioned in Section 2 elimination to h-propositions from inductive types in `\Prop` does not involve constructors with parameters of the type `I`. This means that such constructors can be omitted in definitions of functions by pattern matching in this case. Eliminator corresponding to the recursion principle for `nonempty` can be defined using pattern matching as follows:

   ```
   \lemma rec {A B : \Type} (p : isProp B) (t : nonempty A) (f : A -> B) : B \elim t
         | in a => f a
   ```

   This simplified pattern matching allows to derive proof irrelevance for types in `\Prop` using `nonempty`:

   ```
   \lemma prop-pi {A : \Prop} {a a' : A} : a = a'
     => \case trunc (in a) (in a') __ \with { -- \case f __ is unfolded to
       | in x => x                             -- \lam i => \case f i
     }
   ```

2. **Truncated inductive types**. Any definition of an inductive type can be placed in a universe of any given homotopy level by marking the definition as *truncated*. This marking is done by means of the keyword `\truncated` in front of a definition. It places the definition in a universe `\Type n k`, but at the cost of restricting elimination to types `A` satisfying `isOfHLevel A k`. In other words, pattern matching cannot be used to define functions from truncated inductive types in `\Type n k` to a type `A` unless there is a proof of `isOfHLevel A k`. For example, the quotient set can be defined as follows:

   ```
   \truncated \data Quotient {A : \Type} (R : A -> A -> \Type) : \Set
     | in~ A
     | ~-equiv (x y : A) (R x y) : in~ x = in~ y -- again, this is unfolded to
                                            -- ~-equiv (x y : A) (R x y) (i : I) with conditions
   ```

   The recursion principle for `Quotient` would be:

```
\sfunc rec {A B : \Type} (R : A -> A -> \Type) (p : isSet B)
           (f : A -> B) (f-R : \Pi (x y : A) (R x y) -> f x = f y) (x : Quotient R) : B
   \elim x
      | in~ a => f a
      | ~-equiv a a' r i => f-R a a' r @ i
```

3.2 Typeclasses and records

Here we highlight some aspects of the system of typeclasses and records in Arend. This system is built upon the system of records in the core type theory discussed in Section 2.

1. **Partial implementations.** As described in Section 2 the mechanism of manifest fields allows to partially implement parent records or classes[11] inside a child record or class. For example, as mentioned in Section 2.3.5 the type of rings can be defined in Arend as an extension of the type of semirings using partial implementation:

   ```
   \class Semiring \extends AbMonoid, Monoid {
     | ldistr {x y z : E} : x * (y + z) = x * y + x * z
     | rdistr {x y z : E} : (x + y) * z = x * z + y * z
     | zro_*-left {x : E} : zro * x = zro
     | zro_*-right {x : E} : x * zro = zro
   }

   \class Ring \extends Semiring, AbGroup {
     | zro_*-left {x} => {?} -- proof of 0 * x = 0 using invertibility wrt '+'
     | zro_*-right {x} => {?} -- proof of x * 0 = 0 using invertibility wrt '+'
   }
   ```

   Here `Semiring` extends `AbMonoid` and `Monoid` which are additive commutative and multiplicative monoidal structures on the same carrier `E`[12]. Since `AbGroup` extends `AbMonoid` with the group structure, the type `Ring` is defined as a semiring where `+` operation is strengthened to a group operation.

   Another example: tight apartness relation `x # y`. It is constructively better behaved variant of inequality `Not (x = y)` since it satisfies the tightness condition `Not (x # y) -> x = y` (whereas `Not (x = y)` constructively does not). Such relations are useful, for example, for defining domains constructively as rings satisfying `x # 0 -> y # 0 -> (x * y) # 0`. It is natural to define the base class `Set#` of sets with apartness relation and various types of structures with apartness as its extensions. However, for example, in case of groups specifying the full binary relation `x # y` is redundant since it can be expressed in terms of `x # 0`. This can be done using partial implementations:

   ```
   -- groups with '+' operation (not necessarily commutative) and apartness relation '#'
   \class AddGroupWith# \extends AddGroup, Set_#
       | \fix 8 #0 : E -> \Prop
       | #0-zro : Not (zro `#0)
       | #0-negative {x : E} : x `#0 -> negative x `#0
       | #0-+ {x y : E} : (x + y) `#0 -> x `#0 || y `#0
       | #0-tight {x : E} : Not (x `#0) -> x = zro

       | # x y => (x - y) `#0
       -- we omit implementations of properties of x # y
   ```

   Finally, another example from Section 2.3.5 - the type of Heyting fields which is by definition a commutative local ring where all noninvertible elements are equal to zero. This example illustrates

[11]The difference between classes and records is not relevant at this point. We discuss specifics of classes in more detail below
[12]In arend-lib there are two hierarchies for groups and monoids: one for + and one for ∗.

the usefulness of partial implementations for defining structures which extend several branches of hierarchies in such a way that some data and properties of one branch is implemented in terms of data and properties of other branches.

```
\class Field \extends LocalCRing, GCDDomain
  | #0 x => Inv x
  | #0-+ => LocalRing.sumInv=>eitherInv -- property of local rings: if x + y is invertible
                                        -- then either x or y is invertible.
  -- we omit implementations of isGCDDomain and other properties of apartness '#'

  -- the tightness '#0-tight' remains unimplemented and represents one of defining
  -- properties of the Heyting field: all noninvertible elements are equal to zero
```

2. **Anonymous extensions.** Extensions of records and classes can be defined on the fly: if `C` is a record containing a field `f : A` then the term `C { f => a }` refers to a record extension obtained from `C` by specifying `f` to a term `a : A`.

   For example, one can use the class of all semirings to define the type of all semiring structures on natural numbers:

   ```
   \func SemiringsOnNat : \Set => Semiring { E => Nat } -- or simply '=> Semiring Nat'
   ```

   Here `SemiringsOnNat` is a subtype of `Semiring`.

3. **Properties.** As mentioned in Section 2 a form of computational proof irrelevance holds in Arend for records and classes. Arend makes an explicit distinction between property fields and data fields. The result of comparison of two objects depends only on comparison of data fields, proofs of properties are ignored. By default a field is a property if its type is a proposition which in particular is the case if the type is equality between elements of a set.

   For example, proofs of properties `ldistr`, `rdistr`, `zro_*-left` and `zro_*-right` of the class `Semiring` are ignored when comparing two semirings.

4. **Static and dynamic blocks.** All fields of a record or class `D` as well as definitions of inductive types and functions declared in the inner block of `D` have an implicit parameter `\this : D` which represents the instance [13]. Fields and definitions in this block are thus *dynamic* analogously to dynamic class methods in Java or C++. Although dynamic definitions can be replaced with external static definitions with an extra parameter of type `D`, dynamic definitions of basic operations and properties of `D` are more concise and neat. For example, part of dynamic block of `Monoid` might look as follows:

   ```
   \class Monoid \extends Pointed {
     | \infixl 7 * : E -> E -> E
     | ide-left {x : E} : ide * x = x
     | ide-right {x : E} : x * ide = x
     | *-assoc {x y z : E} : (x * y) * z = x * (y * z)

     \func pow (a : E) (n : Nat) : E \elim n
       | 0 => ide
       | suc n => pow a n * a

     \lemma pow_+ {a : E} {n m : Nat} : pow a (n + m) = pow a n * pow a m \elim m
       | 0 => inv ide-right
       | suc m => pmap (`* a) pow_+ *> *-assoc
   }
   ```

[13]Expressions $a.f$ for taking value of a field or invoking a definition f of an instance a are converted to applications $f\,\{a\}$

Just like any other definition in Arend classes and records have `\where` block for associated *static* definitions. These static blocks are quite useful, for example, for defining some standard structures on top of a mathematical structure like having an apartness relation or decidable equality. For example, for rings one use the static block as follows:

```
\class Ring \extends Semiring, AbGroup {
 -- ...
} \where {
-- | A ring with a tight apartness relation.
  \class With# \extends Ring, AddGroup.With# {
      -- ...
  }

  -- | A ring with decidable equality.
  \class Dec \extends AddGroup.Dec, With# {
      -- ...
  }
}
```

5. **Type classes.** The concept of a type class originates in Haskell programming language. The mechanism of type classes enables ad-hoc polymorphism, a form of polymorphism similar to operator overloading in object-oriented programming: type classes specify bundles of operations on types which can be implemented differently for different types.

   In Arend type classes are definitions which start with the keyword `\class` and are similar to records. The key difference is the *instance inference* mechanism for type classes: some of objects of a type class can be declared as instances which would enable the typechecker to infer a suitable instances whenever necessary. For example, a semiring structure on natural numbers can be defined as an instance of class `Semiring`:

```
\instance NatSemiring : Semiring Nat
  | zro => 0
  | + => +
  | * => *
  | ide => 1
  -- we omit proofs of semiring properties for Nat
```

   At a place where multiplication of natural numbers is used typechecker can infer the instance `NatSemiring` and make applicable at this place everything that is applicable to semirings, for example, a simplification tactic erasing subexpressions multiplied by 0.

```
\func example (n : Nat) : n * 0 + 1 = 1
  => simplify -- the tactic 'simplify' infers NatSemiring and uses proofs
              -- of properties of +, * to derive the equality
```

   With respect to the behavior of instances and instance inference Arend is very similar to other theorem provers like Coq, Lean or Agda.

3.3 Arrays

The type Array is defined in Prelude.ard as a record:

```
\record DArray {len : Nat} (A : Fin len -> \Type)
              (\coerce at : \Pi (j : Fin len) -> A j)
\func Array (A : \Type) => DArray { | A _ => A }
```

The are also functions `nil` and `a :: l` which can be used in pattern matching as if `DArray` were an inductive type. Coercion `at` allows to use arrays as if they were functions. A function `G`, as mentioned in Section 2.3.6, can be defined by recursion on terms as follows:

```
\data Term (F : \Set) (a : F -> Nat) | fun (f : F) (v : Array (Term F a) (a f))
\func G {F : \Set} {a : F -> Nat} (t : Term F a) \elim t
  | fun f v => fun f (\lam i => G (v i))
```

4 The library arend-lib

In this section we highlight some parts of arend-lib.

4.1 Constructive mathematics

4.1.1 Category theory

4.1.2 Topology

Locales. The concept of *a locale* captures the system of open subsets $O(X)$ of a topological space X without reference to its underlying set of points. Locales align better with constructive mathematics, as many natural systems of "open subsets" lack enough points without the axiom of choice. For instance, classically, an affine scheme is defined as the topological space $\mathsf{Spec}\,R$, whose points are the prime ideals of a ring R, equipped with a canonical sheaf of rings. However, in a constructive setting, prime ideals may not exist. In contrast, the locale of "open subsets" corresponding to radical ideals of R always preserves the meaningful structure. Therefore, it is more natural in constructive mathematics to define an affine scheme as this locale of radical ideals, together with its sheaf of rings. There are also reasons to use locales instead of topological spaces in the classical setting, we will discuss this in the context of algebraic geometry in Section 4.1.3.

Formally, $O(X)$ is defined as a *frame*, which is a poset with all small joins $\vee$ and all finite meets $\wedge$ satisfying the infinite distributive law: $x \wedge (\vee_i\, y_i) \leq \vee_i\, x \wedge y_i$. In particular, a frame is a complete, distributive lattice. A *frame homomorphism* is a homomorphism of posets preserving finite meets and small joins. This defines the category Frm of frames. The category Locale of locales is the dual of Frm: $\mathsf{Locale} := \mathsf{Frm}^{\mathrm{op}}$. In particular, a locale is given by the same data as a frame.

In arend-lib the type of locales is defined as the class `Locale (E : \Set)` extending classes `CompleteLattice`, `Bounded.DistributiveLattice` and `SiteWithBasis` (sites are pairs $(\mathcal{C}, J)$ where $\mathcal{C}$ is a category with a covering J for objects). The class `Locale` forms a diamond in lattice hierarchy: the class `CompleteLattice` is a descendant of `Bounded.Lattice` extending it with operation `Join {J : \Set} : (J -> E) -> E` for arbitrary joins $\vee_{j\in J}\, x_j$ and the class `Bounded.DistributiveLattice` extends `Bounded.Lattice` with finite distributivity `ldistr>= {x y z : E} : x ∧ (y ∨ z) <= (x ∧ y) ∨ (x ∧ z)`.

The categories of frames and locales are defined as follows:

```
\record FrameHom \extends SetHom {
  \override Dom : Locale
  \override Cod : Locale
  -- Properties expressing that 'fun', the field of SetHom, is a homomorphism of frames
  | fun-top : fun top = top
  | fun-meet {x y : Dom} : fun (x ∧ y) = fun x ∧ fun y
  | fun-Join {J : \Set} {f : J -> Dom} : fun (Join f) = Join (\lam j => fun (f j))

  -- ...
}

\func FrameCat : Cat Locale \cowith
  | Hom => FrameHom
  | id L => \new FrameHom {
    | fun x => x
    | fun-top => idp -- proof by reflexivity
    | fun-meet => idp
    | fun-Join => idp
  }
  | o {L M K : Locale} (g : FrameHom M K) (f : FrameHom L M) : FrameHom L K => \new FrameHom {
    | fun x => g (f x) -- f and g are coerced to functions
    | fun-top => pmap g fun-top *> fun-top
    -- *> stands for composition of paths
    -- 'pmap' maps path between args to path between values of 'g'
    | fun-meet {x} {y} => pmap g fun-meet *> fun-meet
    | fun-Join {J} {h} => pmap g fun-Join *> fun-Join
  }
  -- Proofs that this defines a category
  -- ...

\func FrameBicat : BicompleteCat \cowith
  | Cat => FrameCat
  | limit G =>  \new Limit {
    | apex => limit-obj G
    -- ...
    }
  | pullback {X} {Y} {Z} f g => \new Pullback {
    | apex => pullback-obj f g
    -- ...
    }
  | colimit G => reflectiveSubPrecatColimit FrameUnitalReflectiveSubcat G
    (FrameUPresCocompleteCat.colimit (Comp FrameUnitalReflectiveSubcat G))
\where {
  \func limit-obj {J : SmallPrecat} (G : Functor J FrameCat) : Locale \cowith
    | E => \Sigma (P : \Pi (j : J) -> G j) (\Pi {j j' : J} (h : Hom j j') -> G.Func h (P j) = P j')
    | <= (P,_) (Q,_) => \Pi (j : J) -> P j <= Q j
    -- ...

  \func pullback-obj {L M K : Locale} (f : FrameHom L K) (g : FrameHom M K) : Locale \cowith
      | E => \Sigma (x : L) (y : M) (f x = g y)
      | <= P Q => \Sigma (P.1 <= Q.1) (P.2 <= Q.2)
      -- ...
}

-- Declaring 'LocaleCat' an \instance (as opposed to \func in case of 'FrameCat')
-- enables instance inference mechanism to infer 'LocaleCat' instance for locales
\instance LocaleCat : BicompleteCat \cowith
  | Cat => Cat.op {FrameBicat}
  | limit (G : Functor) => FrameBicat.colimit G.op
  | colimit (G : Functor) => FrameBicat.limit G.op
  | terminal {
    | apex => discrete (\Sigma) -- Locale of subobjects of the unit type
    -- ...
  }
```

4.1.3 Algebra

Algebraic geometry. The part of library arend-lib devoted to algebraic geometry contains basics of the theory of *schemes*. The constructive theory of schemes developed in arend-lib is based on locales, which are better suited for constructive algebraic geometry than topological spaces as explained in Section 4.1.2.

Schemes in arend-lib are defined in terms of locally ringed locales as follows:

1. A *ringed locale* is a locale L together with a ring-valued sheaf $\mathcal{O}_L$ on it:

```
\record RingedLocale (\coerce L : Locale) (R : VSheaf CRingCat L)
-- Here 'VSheaf D C' is the type of D-valued sheaves on a site C (i.e. on a category
-- with a covering for objects). Note that 'Locale' is a subtype of 'Site'.
```

2. Standardly, locally ringed spaces are defined as ringed spaces $(X, \mathcal{O}_X)$ such that the stalk $\mathcal{O}_{X,x}$ at every point $x \in X$ is a local ring. This definition is not suitable for locales since it refers to points and this does not work constructively as explained in Section 4.1.2. Nevertheless, a *locally ringed locale* can be defined by using an alternative formulation of the locality condition which is classically equivalent to the standard one:

```
\record LocallyRingedLocale \extends RingedLocale
  | isNonTrivial (a : L) : 0 = {R.F a} 1 -> a <= bottom
  | isLocallyRinged (a : L) (x : R.F a) : a <= SJoin (\lam b => \Sigma (p : b <= a)
    (EitherInv \this p x))
  -- Here \func SJoin (U : E -> \Prop) => Join (\lam (t : Total U) => t.1)
  \where {
    \func EitherInv (L : RingedLocale) {a b : L} (p : b <= a) (x : R.F a)
      => Inv (R.F.Func p x) || Inv (R.F.Func p x + 1)
  }
```

3. A *scheme* is a locally ringed locale which is locally isomorphic (by univalence this is the same as equal) to an *affine* locally ringed locale (which classically is the spectrum $\mathsf{Spec}\, R$ of a ring R):

```
\record Scheme \extends LocallyRingedLocale
  | isScheme : L.top <= SJoin (\lam U => ∃ (R' : CRing) (VSheaf.restrict U R = {RingedLocale}
    affineRingedLocale R'))
  | isNonTrivial a p => -- ...
  | isLocallyRinged a x => -- ...
```

Affine locally ringed locales are constructed using presentations (`RingedFramePres`, `LocallyRingedFramePres`). It is easier to construct sheaves on presented frames:

```
\func Spec (R : CRing) : Locale
  => PresentedFrame (SpecPres R)

\func affineRingedLocale (R : CRing) => locallyRingedLocaleFromPres (affineRingedPres R)
```

The definition of `affineScheme` uses univalence:

```
\func affineScheme (R : CRing) : Scheme \cowith
  | RingedLocale => affineRingedLocale R
  | isScheme {x} _ => -- proof using univalence
```

PID domains. As we mentioned earlier (Section 3.2), domains are defined constructively as rings with apartness relation `x # y` satisfying `x # 0 -> y # 0 -> (x * y) # 0`. According to the standard classical definition a domain R is a *Principal Ideal Domain* (PID) if *every* ideal $I \subseteq R$ is *principal*, that is there exist $a \in R$ such that I is generated by a: $I = (a)$. Constructively, this definition is too strong since existence of generators a for ideals I even for the most basic classical examples can only be proven using AC or LEM. For example, for every predicate $P : \mathbb{Z} \to \mathsf{Prop}$ the ideal generated by $\{0\} \cup \{x \in \mathbb{Z} \,|\, P(x)\}$ is principal if and only if P is *decidable* which means that $\prod_{x:\mathbb{Z}} P(x) \vee \neg P(x)$ is provable. Since not every predicate on $\mathbb{Z}$ is decidable constructively, $\mathbb{Z}$ is not PID according to the classical definition.

A constructively more well-behaved property is the requirement that every *finitely generated* ideal be principal. This is equivalent to the *Bezout condition*, which is defined in arend-lib as follows:

```
\func IsBezout => \Pi (a b : E) -> ∃ (s t : E) (LDiv (s * a + t * b) a) (LDiv (s * a + t * b) b)

\class BezoutRing \extends CRing, GCDMonoid {
  | isBezout : IsBezout
  | isGCD a b => -- proof of TruncP (GCD a b)
  | gcd-ldistr c {a} {b} {z} (g : GCD a b z) => -- proof of TruncP (GCD (c * a) (c * b) (c * z))
}
```

were `LDiv x y` is the type of left divisors and `GCD` is the type of greatest common divisors:

```
\record DivBase {M : Monoid} (\coerce val : M) (elem inv : M)

\record LDiv \extends DivBase
    | inv-right : val * inv = elem

\record GCD {M : CMonoid}
            (val1 val2 : M) (\coerce gcd : M)
            (gcd|val1 : LDiv gcd val1)
            (gcd|val2 : LDiv gcd val2)
            (gcd-univ : \Pi (g : M) -> LDiv g val1 -> LDiv g val2 -> LDiv g gcd)
```

Now, if we assume LEM, then the classical definition of PID can be equivalently reformulated as follows: PID is a domain satisfying Bezout condition and such that all ideals are finitely generated. The latter condition is one of equivalent definitions of *Noetherian property* assuming LEM. Constructively, however, this condition is too strong. Instead, we say that a ring R is Noetherian if for every sequence $I_1 \subseteq I_2 \subseteq \dots$ of finitely generated ideals of R there is $k \in \mathbb{N}$ such that $I_k = I_{k+1}$.

```
\class NoetherianCRing \extends CRing
  | isNoetherian (I : Nat -> Ideal \this) : (\Pi (n : Nat) -> Ideal.IsFinitelyGenerated {I n}) ->
    Ideal.ChainCondition I

-- function accompanying the class 'Ideal'
\func ChainCondition {R : CRing} (I : Nat -> Ideal R) =>
    (\Pi (n : Nat) {a : R} -> I n a -> I (suc n) a) -> ∃ (n : Nat) ∀ {a} (I (suc n) a -> I n a)
```

In arend-lib PIDs are defined as domains R satisfying Bezout condition, Noetherian property and such that equality $x =_R y$ is a decidable predicate. For Bezout rings Noetherian property admits an equivalent formulation which is more convenient to work with. To state it, consider equivalence relation $x \sim y := x \,|\, y \wedge y \,|\, x$ on R and consider the quotient monoid $R_\sim$. Then every sequence $\{a_i\} \subset R_\sim$ satisfying $a_{i+1} \,|\, a_i$ necessarily has $a_k = a_{k+1}$ for some $k \in \mathbb{N}$ if and only if R satisfies the Noetherian condition.

The combination of the Bezout condition and the Noetherian property places PIDs in an important class of domains known as *Smith domains*. The distinguishing property of Smith domains is existence of the Smith normal form for matrices over them. Namely, for every $m \times n$, $n \geq m$ matrix A over a Smith domain R there exist invertible matrices S, T over R such that $S \cdot A \cdot T = (diag(a_1, \dots, a_k, 0^{m-k}) \,|\, \mathbf{0}^{m\times(n-m)})$ is $m \times m$ diagonal matrix followed by $m \times (n-m)$ zero matrix and $a_i \,|\, a_{i+1}$. In other words, if we say $A \sim B$ for matrices if there exist invertible S, T such that $B = S \cdot A \cdot T$, then for every A there exist a matrix in the normal form D such that $A \sim D$. Smith domains can be equivalently defined as domains satisfying the *Kaplansky condition*.

```
\type IsKaplansky (R : CRing) => \Pi (a b c : R) -> IsCoprimeArray (a,b,c) -> ∃ (t s : R)
    (IsCoprime (t * a) (t * b + s * c))

\class SmithRing \extends StrictBezoutRing
  | isKaplansky : IsKaplansky \this

\class SmithDomain \extends BezoutDomain.Dec, SmithRing
```

The Smith normal form theorem is proven as part of the equivalence:

```
\lemma Smith-char {R : CRing} : TFAE (
-- The four conditions below are equivalent
    \Sigma R.IsStrictBezout (IsKaplansky R),
    \Sigma R.IsStrictBezout (\Pi (A : Matrix R 2 2) -> A 1 0 = 0 -> IsCoprimeArray (A 0 0, A 0 1,
    A 1 1) -> ∃ (B : Matrix R 2 2) (IsDiagonal B) (A M~ B)),
     -- here M~ is equivalence as in the Smith normal form
    \Pi {n m : Nat} -> n <= m -> m <= 2 -> \Pi (A : Matrix R n m) -> ∃ (B : Matrix R n m)
    (IsDiagonal B) (A M~ B),
    \Pi {n m : Nat} (A : Matrix R n m) -> ∃ (B : Matrix R n m) (IsSmith B) (A M~ B)
  )

-- Dynamic lemma (using \this parameter) in SmithRing class
\lemma toSmith {n m : Nat} (A : Matrix E n m) : ∃ (B : Matrix E n m) (IsSmith B) (A M~ B)
    => Smith-char 0 3 (isStrictBezout,isKaplansky) A
```

Since PIDs are Smith domains, in arend-lib they are defined as extension:

```
\class PID \extends SmithDomain, NoetherianCRing {
  | divChain : CMonoid.DivChain {DivQuotientMonoid nonZeroMonoid}
  | isNoetherian I Ifg => Ideal.fromMonoidChainCondition (IntegralDomain.Dec.fromNonZeroDivChain
    divChain) I \lam n => bezout_finitelyGenerated_principal.1 isBezout (Ifg n) -- f.g. ideals
--  are principal in Bezout rings
  | isKaplansky => adequate_kaplansky \lam a/=0 => adequate'_adequate (domain_adequate' divChain
    a/=0)
}
```

The proof of Kaplansky condition goes via the proof of *adequate condition* for PIDs first.
Among the properties of PIDs proven in arend-lib, for example, is that PIDs are 1-dimensional:

```
\lemma is1Dim : Dim<= 1
    => bezout_1Dim<->adequate'.2 \lam a/=0 => domain_adequate' divChain a/=0

-- Dynamic type (using \this parameter) in CRing class
\type Dim<= (k : Nat) => \Pi (x : Array E (suc k)) ->
                          ∃ (a : Array E (suc k)) (m : Array Nat (suc k)) (fold x a m = 0)
    \where
      \func fold {k : Nat} (xs as : Array E k) (ms : Array Nat k) : E \elim k, xs, as, ms
        | 0, nil, nil, nil => 1
        | suc k, x :: xs, a :: as, m :: ms => pow x m * (fold xs as ms - x * a)
```

Polynomials of one or several variables. Polynomials of one variable over any pointed type (a type with 0) have two equivalent representations in arend-lib: as an inductive type with conditions `Poly` and as a quotient set `QPoly`. Both are essentially arrays of coefficients quotioned by cutting trailing zeros, but sometimes it is more convenient to work with `Poly` instead of `QPoly` and vice versa.

```
\data Poly (R : AddPointed)
  | pzero
  | padd (Poly R) R
  | peq : padd pzero 0 = pzero

\func QPoly (R : AddPointed) => Quotient {Array R} (__ = __ ++ 0 :: nil)
```

Multivariate polynomials are defined as a monoid algebra:

```
\func MPoly (J : \Set) (R : CRing) => MonoidAlgebra (PermSetMonoid J) R

\instance PermSetMonoid (A : \Set) : CMonoid (PermSet A)
```

```
\type PermSet (A : \Set) => Quotient {Array A} EPerm
-- here the relation EPerm l l' holds iff l' can be obtained by permuting elements in l
```

Matrices over commutative rings. The library arend-lib defines matrices over any type R as two-dimensional arrays and then formalizes operations on matrices and their properties for a variety of structures on R. In case of commutative rings arend-lib has definitions of determinant, characteristic polynomial and proofs of a number of their properties including the proof of Cayley-Hamilton theorem.

```
\func determinant {R : CRing} {n : Nat} (M : Matrix R n n)
  => R.FinSum \lam (e : Sym n) => sign e * R.BigProd (\lam j => M (e j) j)

\func charPoly {R : CRing} {n : Nat} (M : Matrix R n n) : Poly R
  => determinant $ padd 1 0 *c matrix-map (padd 0) 1 - matrix-map (padd 0) M
-- det (x * I - M)
-- 'matrix-map' maps in this case a matrix over 'R' to a matrix over 'Poly R'

\lemma cayley-hamilton {R : CRing} {n : Nat} (A : Matrix R n n)
                : polyMapEval (CAlgebra.coefHom {MatrixAlgebra R n}) (charPoly A) A = 0
    -- 'polyMapEval' converts a polynomial over R to polynomial over 'MatrixAlgebra R n'
    -- and then evaluates it at the matrix A
    => -- ...
```

Among the properties of determinant formalized in arend-lib is the cofactor expansion and the existence of adjugate matrices.

```
\func adjugate {R : CRing} {n : Nat} (M : Matrix R n n) : Matrix R n n \elim n
  | 0 => M
  | suc n => mkMatrix \lam i j => Monoid.pow -1 (j + i) * determinant (minor M j i)

\lemma adjugate-left {R : CRing} {n : Nat} {M : Matrix R n n} : adjugate M * M = determinant M *c 1
  => -- ...

\lemma adjugate-right {R : CRing} {n : Nat} {M : Matrix R n n}
                                        : M * adjugate M = determinant M *c 1
  => -- ...
```

As a corollary we have that M is invertible if and only if $\mathsf{det}(M)$ is invertible.

```
\lemma determinant-inv {R : CRing} {n : Nat} {M : Matrix R n n} : Inv M <-> Inv (determinant M)
```

Another corollary is Nakayama's lemma, which says the following. Let R be a commutative ring, $I \subseteq R$ an ideal and M a finitely generated module over R. If $I \cdot M = M$ then there exist $r \in R$ such that $r \equiv 1 (\mathsf{mod}\, I)$ such that $r \cdot M = 0$.

```
\lemma nakayama {R : CRing} {M : LModule R} {I : Ideal R} (fg : M.IsFinitelyGenerated)
                (c : topSubLModule M <= I *m topSubLModule M)
                : ∃ (e : R) (I (1 - e)) ∀ (m : M) (e *c m = 0)
```

Linear algebra over Smith domains. Here we sketch some elements of linear algebra over Smith domains formalized in arend-lib.

1. Definition of the rank of a matrix over a Smith domain:

   ```
   \func rank {R : SmithDomain} {n m : Nat} (A : Matrix R n m) : Nat => -- ...
   ```

2. Let R be a Smith domain. For any finite-dimensional free modules U, V over R and every linear $f : U \to V$ the image $\mathsf{Im}\, f$ and the kernel $\mathsf{Ker}\, f$ modules are finite-dimensional free.

```
\class FinModule \extends LModule
  | isFinModule : ∃ (l : Array E) (IsBasis l)

\instance image-fin {R : SmithDomain} {U V : FinModule R} (f : LinearMap U V) : FinModule
  | LModule => ImageLModule f
  | isFinModule => \case U.isFinModule, V.isFinModule \with {
    | inP (lu,bu), inP (lv,bv) => TruncP.map (basis f bu bv) \lam s => (s.1,s.2)
  }
\where {
    \protected \lemma basis {U V : LModule R} (f : LinearMap U V) {lu' : Array U}
                      (bu' : U.IsBasis lu') {lv' : Array V} (bv' : V.IsBasis lv')
      : ∃ (l : Array (Image f)) (IsBasis {ImageLModule f} l)
                        (l.len = rank (LinearMap.toMatrix lu' bv' f))
     => -- ...
}

\instance kernel-fin {R : SmithDomain} {U V : FinModule R} (f : LinearMap U V) : FinModule
  | LModule => KerLModule f
  | isFinModule => -- ...
```

3. Let R be a Smith domain and U a finite-dimensional free module over R. Linear dependence for a set of vectors u_1, ..., u_n in U is decidable.

```
\func dependency-dec {R : SmithDomain} {U : FinModule R} (l : Array U)
                                      : Or (U.IsDependent l) (U.IsIndependent l)
```

Algebraic closure of countable, discrete fields. Constructively there are fewer algebraically closed fields and algebraically closed extensions are harder to construct. For example, the field $\mathbb{C}$ is not algebraically closed constructively (the Fundamental Theorem of Algebra fails).

The library arend-lib defines intergral extensions of rings, splitting fields for polynomials over discrete fields and algebraic closure for countable, discrete fields.

```
\func isIntegral (f : RingHom) (x : f.Cod) : \Prop
  => ∃ (p : Poly f.Dom) (isMonic p) (polyEval (polyMap f p) x = 0)
```

A variant of the following fact is proven in arend-lib: if $S \to R$ is an integral extension of rings and S is 0-dimensional then R is 0-dimensional. The 0-dimensionality `Dim<= 0` condition can equivalently be reformulated as follows:

```
-- Dynamic type (using \this parameter) in Ring class
\type IsZeroDimensional => \Pi (a : E) -> ∃ (b : E) (n : Nat) (pow a n = pow a (suc n) * b)
```

The library arend-lib has a proof of a strengthening of this fact with S a discrete field. A discrete field is a commutative ring such that every element is either zero or invertible (see Section 3.2 for a definition of a field).

```
\class DiscreteField \extends Field, EuclideanDomain {
  | finv : E -> E
  | finv_zro : finv 0 = 0
  | finv-right {x : E} (x/=0 : x /= 0) : x * finv x = 1
  \field eitherZeroOrInv (x : E) : (x = 0) || Inv x
}

\func integralExt-zeroDim {K : DiscreteField} {R : CRing} {f : RingHom K R} (fi : isIntegralExt f)
    : R.IsZeroDimensional
=> -- ...
```

The definition of a splitting field for a polynomial:

```
\type IsSplittingField {k K : DiscreteField} (p : Poly k) (f : RingHom k K)
  => ∃ (l : Array K) (c : K) (RingHom.isAlgebraGenerated f l)
  (polyMap f p = c PolyRing.*c Monoid.BigProd (map (\lam a => padd 1 (negative a)) l))
```

A ring K is algebraically closed if every monic polynomial over K of non-zero degree has a root.

```
\type IsAlgebraicallyClosed (K : Ring)
  => \Pi {p : Poly K} -> ∃ (n : Nat) (n /= 0) (degree<= p n) (polyCoef p n = 1) -> ∃ (a : K)
    (polyEval p a = 0)
```

Every countable discrete field has an algebraic closure:

```
\sfunc countableAlgebraicClosure {k : DiscreteField} (c : Countable k)
: \Sigma (K : DiscreteField) (f : RingHom k K) (IsAlgebraicClosure f)
=> -- ...
```

4.2 Other parts of the library

4.2.1 Synthetic homotopy theory

Eckmann-Hilton argument. The algebraic Eckmann-Hilton argument shows that if a type X has two unital binary operations $\circ$ and $\#$ such that $(a\#b) \circ (c\#d) = (a \circ c)\#(b \circ d)$ then $\circ$ and $\#$ coincide and are commutative and associative.

```
\class Algebraic-Eckmann-Hilton (X : \Type) {
  | \infix 7 o : X -> X -> X
  | \infix 7 # : X -> X -> X
  | id_o : X
  | id_o-left {x : X} : id_o o x = x
  | id_o-right {x : X} : x o id_o = x

  | id_# : X
  | id_#-left {x : X} : id_# # x = x
  | id_#-right {x : X} : x # id_# = x

  | rel {a b c d : X} : (a # b) o (c # d) = (a o c) # (b o d)

  \func units-coincide : id_o = id_# => -- ...
  \func binop_rels_1 {a b : X} : a o b = b # a => -- ...
  \func binop_rels_2 {a b : X} : b # a = b o a => -- ...
  \func comm {a b : X} : a o b = b o a => binop_rels_1 *> binop_rels_2
  \func binops_coincide {a b : X} : a o b = a # b => comm *> binop_rels_1 {\this} {b} {a}
  \func comm-# {a b : X} : a # b = b # a => rewrite (inv binops_coincide, binop_rels_1) idp

  \func rel-o {a b c d : X} : (a o b) o (c o d) = (a o c) o (b o d) =>
    rewrite (binops_coincide {\this} {a} {b}, binops_coincide {\this} {c} {d}, rel,
    inv binops_coincide) idp

  \func assoc {a b c : X} : (a o b) o c = a o (b o c) => rewrite (inv $ id_o-left {\this} {c},
    rel-o, id_o-left, id_o-right) idp
}
```

A corollary of this: the 2-dimensional loop space $\Omega^2(X)$ is a commutative monoid with respect to loop composition.

```
\class Omega^2-Commutative (X : HPointed) {
  \func Commutative (alp bet : Omega^ 2 X) : alp *> bet = bet *> alp => -- ...
}
```

Eilenberg-MacLane space $K_1(G)$. The space $K_1(G)$ is defined to be a space such that its only nontrivial homotopy group is the fundamental group π_1 and it is isomorphic to G.

```
\truncated \data K1 (G : Group) : \1-Type
  | base
  | loop G : base = base
  | relation (g g' : G) (i : I) (j : I) \elim i, j {
    | left, j => base
    | right, j => loop g' j
    | i, left => loop g i
    | i, right => loop (g * g') i
  }
\where {
  \func Loop_K1 {G : Group} : (base {G} = base) = G => -- ...
}
```

4.2.2 Computer science

References

[1] Thorsten Altenkirch. Extensional equality in intensional type theory. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, LICS '99. IEEE Computer Society, 1999.

[2] Gilles Barthe, Venanzio Capretta, and Olivier Pons. Setoids in type theory. *Journal of Functional and Logic Programming*, 13:261–293, 2003. Special Issue on Logical Frameworks and Metalanguages.

[3] Jean-philippe Bernardy, Patrik Jansson, and Ross Paterson. Proofs for free: Parametricity for dependent types. *J. Funct. Program.*, 22(2):107–152, March 2012.

[4] Rafaël Bocquet. External univalence for second-order generalized algebraic theories, 2022.

[5] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In *Certified Programs and Proofs: Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, page 147–162, Berlin, Heidelberg, 2013. Springer-Verlag.

[6] Thierry Coquand, Randy Pollack, and Makoto Takeyama. A logical framework with dependently typed records. *Fundam. Inf.*, 2004.

[7] Ulrik Buchholtz Daniel Gratzer, Jonathan Weinberger. Directed univalence in simplicial homotopy type theory. `https://arxiv.org/pdf/2407.09146`.

[8] Kim Nguyen Denis-Charles Cisinski, Bastiaan Cnossen and Tashi Walde. Formalization of higher categories. `https://drive.google.com/file/d/1lKaq7watGGl3xvjqw9qHjm6SDPFJ2-0o/view`.

[9] Cesare Gallozzi. Homotopy type-theoretic interpretations of constructive set theories. July 2018.

[10] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, page 109–138, 1996.

[11] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, page 123–137. Association for Computing Machinery, 1994.

[12] M. Hofmann. *Extensional Concepts in Intensional Type Theory.* Thesis. University of Edinburgh, Department of Computer Science, 1995.

[13] Valery Isaev. A constructive approach to complete spaces, 2024. Available at: `https://arxiv.org/abs/2401.12345`.

[14] Chris Kapulkin and Peter LeFanu Lumsdaine. The simplicial model of univalent foundations (after voevodsky), 2018.

[15] Krzysztof Kapulkin and Peter LeFanu Lumsdaine. Homotopical inverse diagrams in categories with attributes. *Journal of Pure and Applied Algebra*, 2021.

[16] Nikolai Kraus. On the role of semisimplicial types. `https://nicolaikraus.github.io/docs/on_semisimplicial_types.pdf`.

[17] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, page 109–122. Association for Computing Machinery, 1994.

[18] Nuo Li. Quotient types in type theory, July 2015.

[19] H. Lombardi and C. Quitté. *Commutative Algebra: Constructive Methods: Finite Projective Modules.* Algebra and Applications. Springer Netherlands, 2015.

[20] Zhaohui Luo. *Manifest Fields and Module Mechanisms in Intensional Type Theory*, page 237–255. Springer-Verlag, 2009.

[21] Nicolas Magaud and Yves Bertot. Changing data structures in type theory: A study of natural numbers. In *Selected Papers from the International Workshop on Types for Proofs and Programs*, TYPES '00, page 181–196, Berlin, Heidelberg, 2000. Springer-Verlag.

[22] R. Mines, F. Richman, and W. Ruitenburg. *A Course in Constructive Algebra.* 3Island Press, 1987.

[23] John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, 1983.

[24] Egbert Rijke. Introduction to homotopy type theory, 2022.

[25] Claudio Sacerdoti Coen and Enrico Tassi. Working with mathematical structures in type theory. In *Types for Proofs and Programs*, pages 157–172. Springer Berlin Heidelberg, 2008.

[26] BAS SPITTERS and EELIS VAN DER WEEGEN. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21:795–825, 2011.

[27] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. The marriage of univalence and parametricity. 68, 2021.

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

[29] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical agda: a dependently typed programming language with univalence and higher inductive types. *Proc. ACM Program. Lang.*, 2019.

[30] Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, page 347–359, New York, NY, USA, 1989. Association for Computing Machinery.

[31] Tesla Zhang and Valery Isaev. (co)condition hits the path, 2024.

[32] Théo Zimmermann and Hugo Herbelin. Automatic and transparent transfer of theorems along isomorphisms in the coq proof assistant. *CoRR*, abs/1505.05028, 2015.