The Arend Theorem Prover

Fedor Part Valery Isaev Sergey Sinchuk

April 15, 2025

JetBrains Research

Univalent Foundations and Homotopy Type Theory

- *Univalent Foundations* (UF) of mathematics were formulated by Vladimir Voevodsky between 2006 and 2009.

Univalent Foundations and Homotopy Type Theory

- *Univalent Foundations* (UF) of mathematics were formulated by Vladimir Voevodsky between 2006 and 2009.
- Homotopy types are primitives rather than sets.

Univalent Foundations and Homotopy Type Theory

- *Univalent Foundations* (UF) of mathematics were formulated by Vladimir Voevodsky between 2006 and 2009.
- Homotopy types are primitives rather than sets.
- Inspired by the homotopy semantics of MLTT:
  $a =_A b$ – type of paths $a \rightsquigarrow b$,
  $p =_{a =_A b} q$ – type of homotopies $p \rightsquigarrow q$, etc

Univalent Foundations and Homotopy Type Theory

- *Univalent Foundations* (UF) of mathematics were formulated by Vladimir Voevodsky between 2006 and 2009.
- Homotopy types are primitives rather than sets.
- Inspired by the homotopy semantics of MLTT:
  $a =_A b$ – type of paths $a \rightsquigarrow b$,
  $p =_{a =_A b} q$ – type of homotopies $p \rightsquigarrow q$, etc
- *Homotopy Type Theory* (HoTT) extends MLTT with:
  1. Univalence axiom: $(A \cong B) \cong (A =_U B)$.
     (isomorphic structures are equal, U-object classifier)

Univalent Foundations and Homotopy Type Theory

- *Univalent Foundations* (UF) of mathematics were formulated by Vladimir Voevodsky between 2006 and 2009.
- Homotopy types are primitives rather than sets.
- Inspired by the homotopy semantics of MLTT:
  $a =_A b$ – type of paths $a \rightsquigarrow b$,
  $p =_{a =_A b} q$ – type of homotopies $p \rightsquigarrow q$, etc
- *Homotopy Type Theory* (HoTT) extends MLTT with:
  1. Univalence axiom: $(A \cong B) \cong (A =_U B)$.
     (isomorphic structures are equal, U-object classifier)
  2. Higher inductive types: inductive types D + higher path constructors $a =_D b$.
     (quotients, CW complexes)

Synthetic and set-theoretical formalizations in HoTT

1. (Synthetic homotopy theory). Utilizes the equality type to encapsulate complex homotopy structures. E.g. $\mathbb{S}^1$ is HIT with constructor base : $\mathbb{S}^1$ and higher constructor loop : base $=_{\mathbb{S}^1}$ base.
   - Formal proofs are very close to 'textbook' proofs. UA $\Rightarrow \pi_1(\mathbb{S}^1) = \mathbb{Z}$.
   - Has the potential to reshape certain areas of modern mathematics by adopting a higher categorical perspective.
   - Limitations: original HoTT cannot express general ∞-categories (types are ∞-groupoids). We need a *directed* HoTT (active research area).

Synthetic and set-theoretical formalizations in HoTT

1. (Synthetic homotopy theory). Utilizes the equality type to encapsulate complex homotopy structures. E.g. $\mathbb{S}^1$ is HIT with constructor $\mathsf{base} : \mathbb{S}^1$ and higher constructor $\mathsf{loop} : \mathsf{base} =_{\mathbb{S}^1} \mathsf{base}$.
    - Formal proofs are very close to 'textbook' proofs. UA $\Rightarrow \pi_1(\mathbb{S}^1) = \mathbb{Z}$.
    - Has the potential to reshape certain areas of modern mathematics by adopting a higher categorical perspective.
    - Limitations: original HoTT cannot express general ∞-categories (types are ∞-groupoids). We need a *directed* HoTT (active research area).
2. (Set theory). A variant of set theory is a fragment of HoTT.
    - h-propositions – types A s.t. $\mathsf{PI}(A)$, h-sets – types A s.t. $\mathsf{UIP}(A)$.
    - Extensionality: quotients, function and proposition extensionality, structure identity principle.

HoTT/UF in ITPs

Available options wrt ITPs for HoTT/UF:

1. Purely axiomatic HoTT in mainstream ITPs based on MLTT/CIC (Coq, Agda, ~~Lean~~, etc).
    - Badly impairs computational content of MLTT.
    - Formalization quickly becomes unnecessarily cumbersome.

HoTT/UF in ITPs

Available options wrt ITPs for HoTT/UF:

1. Purely axiomatic HoTT in mainstream ITPs based on MLTT/CIC (Coq, Agda, ~~Lean~~, etc).
   - Badly impairs computational content of MLTT.
   - Formalization quickly becomes unnecessarily cumbersome.
2. Fully computational ITPs based on Cubical Type Theories (e.g. Cubical Agda).
   - CTTs are sophisticated two-level theories with interval *pretype* $\mathbb{I}$. Inspiration – constructive models of HoTT in cubical sets $\mathrm{Set}^{\square^{\mathrm{op}}}$.
   - Although CTTs are of theoretical interest, something much simpler might be sufficient for practical formalization.

HoTT/UF in ITPs

Available options wrt ITPs for HoTT/UF:

1. Purely axiomatic HoTT in mainstream ITPs based on MLTT/CIC (Coq, Agda, ~~Lean~~, etc).
   - Badly impairs computational content of MLTT.
   - Formalization quickly becomes unnecessarily cumbersome.
2. Fully computational ITPs based on Cubical Type Theories (e.g. Cubical Agda).
   - CTTs are sophisticated two-level theories with interval *pretype* $\mathbb{I}$. Inspiration – constructive models of HoTT in cubical sets $\mathrm{Set}^{\square^{\mathrm{op}}}$.
   - Although CTTs are of theoretical interest, something much simpler might be sufficient for practical formalization.
3. Pragmatic approach: ITP which focuses on *practical* aspects of HoTT/UF formalization (Arend).

Arend at a Glance

- The development of Arend started in 2015 at JetBrains Research.

Arend at a Glance

- The development of Arend started in 2015 at JetBrains Research.
- The type theory of Arend is based on a mild extension of MLTT with the primitive interval type $\mathbb{I}$ (HoTT-I).
  - With $\mathbb{I}$, HITs are defined as ordinary **inductive types with conditions**.

Arend at a Glance

- The development of Arend started in 2015 at JetBrains Research.
- The type theory of Arend is based on a mild extension of MLTT with the primitive interval type $\mathbb{I}$ (HoTT-I).
  - With $\mathbb{I}$, HITs are defined as ordinary **inductive types with conditions**.
  - Arend supports a powerful system of records and classes with **partial implementations** and **anonymous extensions**.

Arend at a Glance

- The development of Arend started in 2015 at JetBrains Research.
- The type theory of Arend is based on a mild extension of MLTT with the primitive interval type $\mathbb{I}$ (HoTT-I).
  - With $\mathbb{I}$, HITs are defined as ordinary **inductive types with conditions**.
  - Arend supports a powerful system of records and classes with **partial implementations** and **anonymous extensions**.
  - Arend has built-in type for **arrays** ($=$ vectors$+$lists$+(\{0,\ldots,n\} \to A)$).

Arend at a Glance

- The development of Arend started in 2015 at JetBrains Research.
- The type theory of Arend is based on a mild extension of MLTT with the primitive interval type $\mathbb{I}$ (HoTT-I).
  - With $\mathbb{I}$, HITs are defined as ordinary **inductive types with conditions**.
  - Arend supports a powerful system of records and classes with **partial implementations** and **anonymous extensions**.
  - Arend has built-in type for **arrays** ($=$ vectors+lists+($\{0, \ldots, n\} \to A$)).
- A rich tooling for Arend is provided by a plugin for IntelliJ IDEA.

Arend at a Glance

- The development of Arend started in 2015 at JetBrains Research.
- The type theory of Arend is based on a mild extension of MLTT with the primitive interval type $\mathbb{I}$ (HoTT-I).
  - With $\mathbb{I}$, HITs are defined as ordinary **inductive types with conditions**.
  - Arend supports a powerful system of records and classes with **partial implementations** and **anonymous extensions**.
  - Arend has built-in type for **arrays** ($=$ vectors$+$lists$+(\{0, \ldots, n\} \to A)$).
- A rich tooling for Arend is provided by a plugin for IntelliJ IDEA.
- Arend is fully constructive. The main library arend-lib: constructive mathematics (the largest part), synthetic homotopy theory, theoretical computer science.

Arend at a Glance

- The development of Arend started in 2015 at JetBrains Research.
- The type theory of Arend is based on a mild extension of MLTT with the primitive interval type $\mathbb{I}$ (HoTT-I).
  - With $\mathbb{I}$, HITs are defined as ordinary **inductive types with conditions**.
  - Arend supports a powerful system of records and classes with **partial implementations** and **anonymous extensions**.
  - Arend has built-in type for **arrays** (= vectors+lists+($\{0, \ldots, n\} \to A$)).
- A rich tooling for Arend is provided by a plugin for IntelliJ IDEA.
- Arend is fully constructive. The main library arend-lib: constructive mathematics (the largest part), synthetic homotopy theory, theoretical computer science.
- Near future: support for a version of **directed HoTT**.

HoTT-I (Prelude.ard, the interval)

- The interval type:

```
\data I | left | right
```

HoTT-I (Prelude.ard, the interval)

- The interval type:

```
\data I | left | right
```

- Operation `coe` for transport between fibers over `I` (eliminator for `I`):

```
\func coe (A : I -> \Type) (a : A left) (i : I) : A i
  \elim i -- pattern matching on i
          -- just in Prelude.ard (not allowed in general)
  | left => a
```

HoTT-I (Prelude.ard, the interval)

- The interval type:

```
\data I | left | right
```

- Operation `coe` for transport between fibers over `I` (eliminator for `I`):

```
\func coe (A : I -> \Type) (a : A left) (i : I) : A i
  \elim i -- pattern matching on i
          -- just in Prelude.ard (not allowed in general)
  | left => a
```

- The (almost) only computational rules for `coe`:

```
coe A a left => a
coe (\lam i => B) a j => a -- if i is not in FV(B)
```

HoTT-I (Prelude.ard, the path/equality type)

- The path/equality type, the type of functions `I -> A` with fixed endpoints:

```
\data Path (A : I -> \Type) (a : A left) (a' : A right)
  | path (\Pi (i : I) -> A i)
-- in 'path f', 'f left/right' must eval to a/a'

-- infix version for non-dependent A
\func \infix 1 = {A : \Type} (a a' : A)
  => Path (\lam _ => A) a a'
```

HoTT-I (Prelude.ard, the path/equality type)

- The path/equality type, the type of functions `I -> A` with fixed endpoints:

```
\data Path (A : I -> \Type) (a : A left) (a' : A right)
  | path (\Pi (i : I) -> A i)
-- in 'path f', 'f left/right' must eval to a/a'

-- infix version for non-dependent A
\func \infix 1 = {A : \Type} (a a' : A)
  => Path (\lam _ => A) a a'
```

- Taking a point on a path at `i : I`:

```
\func \infixl 9 @ {A : I -> \Type} {a : A left}
{a' : A right} (p : Path A a a') (i : I) : A i \elim p, i
  | path f, i => f i
  | _, left => a | _, right => a'
```

HoTT-I (Prelude.ard, univalence)

- The univalence axiom represented by function `iso`:

```
\func iso {A B : \Type} (f : A -> B) (g : B -> A)
  (p : \Pi (x : A) -> g (f x) = x)
  (q : \Pi (y : B) -> f (g y) = y) (i : I) : \Type
  \elim i
    | left => A
    | right => B
```

HoTT-I (Prelude.ard, univalence)

- The univalence axiom represented by function `iso`:

```
\func iso {A B : \Type} (f : A -> B) (g : B -> A)
  (p : \Pi (x : A) -> g (f x) = x)
  (q : \Pi (y : B) -> f (g y) = y) (i : I) : \Type
  \elim i
    | left => A
    | right => B
```

- Computational rule for `coe`:

```
coe (\lam i => iso A B f g p q i) a0 right => f a0
-- if i not in FV(A, B, f, g, p, q)
```

Examples

- Reflexivity of equality (Prelude.ard):

```
\cons idp {A : \Type} {a : A} => path (\lam _ => a)
```

Examples

- Reflexivity of equality (Prelude.ard):

```
\cons idp {A : \Type} {a : A} => path (\lam _ => a)
```

- Function extensionality, `pmap` and `transport`:

```
\func funext {A : \Type} (B : A -> \Type)
  (f g : \Pi (x : A) -> B x)
  (p : \Pi (x : A) -> f x = g x) : f = g
    => path(\lam i => \lam x => p @ i)
```

Examples

- Reflexivity of equality (Prelude.ard):

```
\cons idp {A : \Type} {a : A} => path (\lam _ => a)
```

- Function extensionality, `pmap` and `transport`:

```
\func funext {A : \Type} (B : A -> \Type)
  (f g : \Pi (x : A) -> B x)
  (p : \Pi (x : A) -> f x = g x) : f = g
    => path(\lam i => \lam x => p @ i)

\func pmap {A B : \Type} (f : A -> B) {a a' : A}
 (p : a = a') : f a = f a'
    => path (\lam i => f (p @ i))
```

Examples

- Reflexivity of equality (Prelude.ard):

```
\cons idp {A : \Type} {a : A} => path (\lam _ => a)
```

- Function extensionality, pmap and `transport`:

```
\func funext {A : \Type} (B : A -> \Type)
  (f g : \Pi (x : A) -> B x)
  (p : \Pi (x : A) -> f x = g x) : f = g
    => path(\lam i => \lam x => p @ i)

\func pmap {A B : \Type} (f : A -> B) {a a' : A}
 (p : a = a') : f a = f a'
    => path (\lam i => f (p @ i))

\func transport {A : \Type} (B : A -> \Type)
 {a a' : A} (p : a = a') (b : B a) : B a'
  => coe (\lam i => B (p @ i)) b right
```

Eliminator J

- Eliminator J for `Path A a a'` can be derived using coe:

```
\func J {A : \Type} {a : A}
  (B : \Pi (a' : A) -> a = a' -> \Type)
  (b : B a idp) {a' : A} (p : a = a') : B a' p
   => coe (\lam i => B (p @ i) (psqueeze p i)) b right

\func psqueeze {A : \Type} {a a' : A} (p : a = a') (i : I)
  : a = p @ i => path (p @ I.squeeze i __)
-- \func squeeze (i j : I) : I is from Prelude.ard
```

- Standard computational rules hold for J (this was an issue for CTTs).

Pattern matching on idp

- Arend supports pattern matching on idp (equivalent to J):

```
\func J1 {A : \Type} {a : A}
 (B : \Pi (a' : A) -> a = a' -> \Type)
 (b : B a idp) {a' : A} (p : a = a') : B a' p
 \elim p
   | idp => b
```

- Formalization of the proof of generalized Blakers-Massey theorem in arend-lib would be next to infeasible without it.

Path algebra

- Basic operations on paths are defined via PM on idp:

```
\func inv {A : \Type} {a a' : A} (p : a = a') : a' = a
  \elim p
    | idp => idp
\func \infixr 9 *> {A : \Type} {a a' a'' : A} (p : a = a')
  (q : a' = a'') : a = a'' \elim q
    | idp => p
```

- A number of very useful computational reductions hold:

```
pmap id => id, pmap (f o g) => pmap f o pmap g,
funext (funextInv p) => p, funextInv (funext p) => p
```

Inductive types with conditions

- Example: the type `Int` of integers (Prelude.ard).

```
\data Int
  | \coerce pos Nat
  | neg Nat \with { zero => pos zero } -- cond - PM on args
```

- The condition makes `neg zero` evaluate to `pos zero`. Elimination/PM must respect this.

Inductive types with conditions

- Example: the type `Int` of integers (Prelude.ard).

```
\data Int
  | \coerce pos Nat
  | neg Nat \with { zero => pos zero } -- cond - PM on args
```

- The condition makes `neg zero` evaluate to `pos zero`. Elimination/PM must respect this.
- This allows to define HITs (using `I`):

```
\data Sphere1
  | base1
  | loop (i : I) : Sphere1
    \with { | left => base1 | right => base1 }
```

Truncations

- Propositional truncation `TruncP (A : \Type)` can be defined as a HIT.

```
\data TruncP (A : \Type)
  | inP A
  | truncP (a a' : TruncP A) : a = a' -- syntactic sugar
\where {
   \use \level levelProp {A : \Type} (a a' : TruncP A)
        : a = a' => path (truncP a a')
}
```

Truncations

- Propositional truncation `TruncP (A : \Type)` can be defined as a HIT.

```
\data TruncP (A : \Type)
  | inP A
  | truncP (a a' : TruncP A) : a = a' -- syntactic sugar
\where {
   \use \level levelProp {A : \Type} (a a' : TruncP A)
       : a = a' => path (truncP a a')
}
```

- `\use \level` ensures typing `TruncP (A : \Type) : \Prop`.

Truncations

- Propositional truncation `TruncP (A : \Type)` can be defined as a HIT.

```
\data TruncP (A : \Type)
  | inP A
  | truncP (a a' : TruncP A) : a = a' -- syntactic sugar
\where {
    \use \level levelProp {A : \Type} (a a' : TruncP A)
        : a = a' => path (truncP a a')
}
```

- `\use \level` ensures typing `TruncP (A : \Type) : \Prop`.
- In Arend truncations can be defined **entirely without HITs**:

```
\truncated \data TruncP (A : \Type) : \Prop
  | inP A -- elimination restricted to h-propositions
```

Universes \Prop and \Set

- Analogous constructs work for `\Set` and universes of higher homotopy level.
- Universes `\Prop` and `\Set` of h-propositions and h-sets explicitly delineate HoTT's set-theoretic fragment in Arend.
- Definitions can be made polymorphic on the homotopy level.

Partial implementations for classes and records

```
\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 { -- AbGroup <- AbMnoid
  | 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.

Another example: tight apartness relation x#y.

- `x # y` is constructively better behaved variant of inequality `Not (x = y)` since it satisfies the tightness condition `Not (x # y) -> x = y`.

Another example: tight apartness relation x#y.

- `x # y` is constructively better behaved variant of inequality `Not (x = y)` since it satisfies the tightness condition `Not (x # y) -> x = y`.
- If we extend the type of sets with relation # with a group structure, the original `x # y` can be implemented using `z # 0`.

```
\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
```

Class system in Arend: summary

- Anonymous extensions can be created on the fly:

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

- Manifest fields `f => a`, used for partial implementations, erase distinction between fields and parameters of classes.
- Partial implementations allow for flexible hierarchies of bundled or semi-bundled definitions.
- Hierarchy Viewer in IDE allows for convenient navigation through hierarchy.

Arrays

- The type `DArray` 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 }
```

- At the same time functions from `DArray` can be defined by pattern matching.

```
\cons nil {A : Fin 0 -> \Type} : DArray A \cowith
  | at => \case __

\cons \infixr 5 :: {n : Nat} {A : Fin (suc n) -> \Type}
  (a : A 0) (l : DArray (\lam j => A (suc j))) : DArray A
\cowith
| at => \case \elim __ \with {
    | 0 => a | suc j => l j }
```

Motivation

- Consider the type of terms $f\ v_1 \dots v_{a(f)}$, where `F` - set of function symbols $\{f\}$, `a` - their arities, `Vector A n` - the standard data type of vectors of length `n`:

```
\data Term (F : \Set) (a : F -> Nat)
  | fun (f : F) (v : Vector (Term F a) (a f))
```

- Assume we want to define a function `G : Term -> Term` by induction on terms, e.g. a substitution, and prove something about it. What kind of elimination/PM do we need?
- For example, in Coq the generated induction principle for `Term` in insufficient. One has to prove a useful one by hand.

Using Array

- This kind of issues are completely resolved by `DArray`/`Array`.

```
\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))
-- 'v' is treated as a function Fun (a f) -> Term F a
```

- If one uses functions `Fun (a f) -> Term F a` instead, one would run into different problems. For example, if `x1` computes to `x2` and `y1` computes to `y2`, than `f x1 y_1` won't compute to `f x2 y_2` (and `Array` and `Vector` have such computational rule).

Overview of arend-lib: constructive mathematics

1. **Algebra.** 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; natural, integer, rational, real and complex numbers and various structures on them.
2. **Topology and analysis.** Topological spaces, locales, uniform spaces, completion of spaces; derivative over topological rings; directed limits for sequences and functions; series and power series.
3. **Category theory.** Categories, functors, adjoint functors, Kan extensions, (co)limits; elementary topoi and Grothendieck topoi.

Overview of arend-lib: synthetic homotopy theory and CS

1. **Synthetic homotopy theory.** Eckmann-Hilton argument; $K_1(G)$; Hopf fibration; localization of universes and modalities; Generalized Blakers-Massey theorem.

   This branch is planned to be revived after directed HoTT language extension is implemented in Arend.

2. **Computer science.** High-order term rewriting systems. Programming Language Foundations in Arend.

Some references

- IntelliJ IDEA plugin features are nicely described (with demonstrations) in Documentation section of Arend site (https://arend-lang.github.io/).
- There is also a link to a paper draft on Arend and arend-lib (some parts are missing, but will be finished soon).
  https://arend-lang.github.io/assets/lang-paper.pdf