Finitary Construction Calculus

Alexander Kuklev[1] ‹a@kuklev.com›
[1]Radboud Univ. Nijmegen, Software Science

1 Bridging the gap beteen type and set theories

Dependent type theories blur the line between types and terms. Every term `x : T` has a type, and many dependent type theories feature types of types called universes so that every type `T` is also term `T : U` in some universe. Let us introduce type theories, where the converse holds, i.e. every term is also a type.

Besides universes, dependent type theories also feature inductive type families, Σ-types, Π-types, and sometimes coinductive type families which subsume Σ- and Π-types (see Basold-Geuvers, 2016). Since inhabitants of universes are already types, we only have to provide typal interpretation for inhabitants of inductive and coinductive type families. To give a concrete an example let us consider the following definitions:

```
data List<T>
  EmptyList<T>
  NonEmptyList<T>(head : T, tail : List<T>)

data InfList<T>(head : T, tail : InfList<T>)
```

Let us allow to use terms `t : Q` of an inductive or coinductive type `Q` on the right hand side of `:` in which case they will be interpreted as subtypes of `Q` comprised of subtrees of `t`. In case of both `t : List<T>` and `t : InfList<T>` the term `t`, if used in a typal context, will be interpreted as a type comprised of t.tail, t.tail.tail, etc. as long as the tails exist. In case of purely inductive (also known as finitary inductive or countable inductive) `Q`, these types are guaranteed to be finite, but otherwise they can be infinite.

This way, every term `n : ℕ` will be identified with the type `Fin n` of natural numbers smaller than `n`, which makes the universe of finite types. In such a type theory it is very convinient to have proper subtyping so we have `0 <: 1 <: ··· <: ℕ` and `0 : 1`, `1 : 2`, etc. Since every inhabitant of every type is now also a type, every type is also a universe of types which are also universes of types in their own right, etc. This way, every type is naturally equipped by a structure of a hereditary multiset.[1] Set-theoretic structure of natural numbers `n : ℕ` is given by von Neumann ordinals `∅`, `{∅, {∅, {∅}}}`, etc.

In case of Aczel's inductive type of multisets $\mathtt{V}_\mathtt{U}$

```
data V
  sup<T : U>(T → V) : V
```

the set-theoretic structure of `m : V` is precisely `m`.

For inhabitants of `V` equivalence of set-theoretic structure implies equivalence of terms, and since it is possible to define unions, separation, etc. for elements of `V`, it is possible to build types with custom wellfounded hereditary set-theoretic structure. Neither of this is true for inhabitants of arbitrary types:

- There are no means to take arbitrary types `X` and `Y` and produce a type with set-theoretic structure equivalent to the union of set-theoretic structures of `X` and `Y`.
- There is also no separation and no replacement for arbitrary types.
- Equivalence of terms $\mathtt{t} =_Q \mathtt{t'}$ is not implied by and the equivalence of the respective types $\mathtt{t} =_U \mathtt{t'}$ or equivalence of their set-theoretic structures.

[1]Beware that equivalence of types `X` and `Y` does not entail that set-theoretic structures coincide.

Set-theoretic structures of inductive types and their inhabitants is always well-founded (well-founded multisets are the same as well-founded preorders), while the set-theoretic strucures of inhabitants of coinductive types are in general not well-founded. For instance, consider

```
def e1 : InfList<Nat>   // e1 = (0, 0, 0, 0,...)
  head ↦ 0
  tail ↦ e1

def e2 : InfList<Nat>   // e1 = (0, 1, 0, 1,...)
  head ↦ 0
  tail ↦ InfList<Nat>
    head ↦ 1
    tail ↦ e2
```

The respective set-theoretic structures are given by the Quine atom `x = {x}` and the Quine atom once removed `x = {{x}}`.

The most striking application of typal interpretation for the inhabitants of inductive types is the course-of-values induction:

$$\frac{\texttt{n : ℕ ⊢ T : Type} \qquad \texttt{· ⊢ base : T 0} \qquad \texttt{n : ℕ, values : ∀(i : n⁺) T(i) ⊢ step : T(n)}}{\texttt{n : ℕ ⊢ ind<T> n base step : T(n)}}$$

Here, when we define the function `f(n)` on natural numbers by induction, the definition of `n`'th value can access all previous values. For type theories at least as strong as the primitive recursive type theory (Buchholtz-Schipp von Branitz, 2024), availability of course-of-values induction is merely a matter of convinience, since it is always possible to define an intermediate function that inductively computes lists of values, and then trim it. Yet, for weaker type theories it becomes essential.

TODO: Discuss cocourse-of-argument coinduction.

2 Elementary recursive type theory

To obtain elementary recursive type theory we will take the primitive recursive type theory `PRTT` recently deveolped by Buchholtz and Schipp von Branitz, and introduce an additional restriction.

`PRTT` features an unusual universe hierarchy

`FinitaryType <: Type <: Type⁺ <: Type⁺⁺ <: ···`

The first universe `FinitaryType` contains `Void`, `Unit` and `Bool`, is closed under forming inductive type families (conjecturally, also quotient inductive type families), Id-types, Σ-types (record types), but not Π-types (function spaces), so the inductive types inside `FinitaryType` remain at most countably infinite.

Using typal interpretations of terms we can also state that it is transitive and Π-subclosed:

$$\frac{\texttt{Γ ⊢ X : FinitaryType} \qquad \texttt{Γ ⊢ x : X}}{\texttt{Γ ⊢ x : FinitaryType}}$$

$$\frac{\texttt{Γ ⊢ X : FinitaryType} \qquad \texttt{Γ ⊢ x : X} \qquad \texttt{Γ, t : x ⊢ Y(t)}}{\texttt{Γ ⊢ Π(t : x) Y(t) : FinitaryType}}$$

Higher universes are assumed to be closed under forming Π-types (function spaces) (conjecturally, arbitrary coinductive type families). The trick to make the theory primitive recursive is to only allow elimination from inductive types to type families inside `FinitaryType`.

We can make the theory elementary recursive by forbidding to capture free variables used for induction by abstraction. For that, we need to introduce "safe" modality for contexts and variables, modify the induction rules to only apply to "safe" variables, and modify the abstraction rule to only work for variables not locked to be "safe" (Beckmann-Weiermann, 2000).

Let us call the resulting theory Elementary Recursive Type Theory `ERTT`. Let us also introduce its extension `ERCC` (Elementary Recursive Construction Calculus) with an impredicative subuniverse of mere propositions `Prop` inside `Type` (but not inside `FinitaryType`) that makes it possible to reason internally about conservativity of over Elementary Recursive Arithmetic `ERA`.

`ERTT` has expressivity to encode the syntax of arithmetic and a proof calculus for classical first order logic as inductive types, and to define its translation into Heyting arithmetic and ultimately Gödel's System `T`, and to prove that both translations are sound. The translations and proofs only require course-of-values elementary recursion and induction respectively.

Of course, it is not possible to construct normalization-by-evaluation algorithm for System `T` using only elementary induction, as it would yield consistency of Peano Arithmetic `PA`, a system much stronger than `ERTT` itself.

However, in Buchholtz-Schipp von Branitz it is sketched, how to naturally encode induction up to ε_0 as a proposition, and use it for a natural relative consistency proof of arithmetic. Strong normalization proof for Gödel's System `T` that was recently formalized in Agda by S. Urciuoli (2023) can be carried out in `ERTT` relative to ε_0-induction, yielding a relative consistency proof of Peano arithmetic.[2]

This exemplifies how `ERTT` can be used to formalize metamathematical results in completely direct and explicit way, omitting the informal, hand-wavy and somewhat error-prone part where arithmetical statements are identified with metamathematical ones through obscure and complicated encodings.

Open Question: can we introduce an inductive-recursive type of codes for inductive types inside `FinitaryType`?

3 Elementary recursive arithmetic (`ERA`)

There is an important class of computable functions on natural numbers called (Kalmár) elementary recursive functions. It has been recently shown (Prunescu et al. 2025) that these functions can be generated using zero (0) and three basic operations: addition ($n + m$), integer remainder ($n \,\%\, m$) and base-two exponentiation (2^n) defined by the following identities:

$$\begin{array}{lll} n + 0 = n & n + (m + 1) = (n + m) + 1 & 2^{n+1} = 2^n + 2^n \\ n \,\%\, 0 = n & (n + m) \,\%\, m = n \,\%\, m & n \,\%\, (n + m) = n \end{array}$$

where 1 is defined as 2^0. Any closed expression composed of these operations eventually reduces to 0, 1, or a sum of 1's by the successive application of these identities as rewrite rules.

Truncated subtraction, exponentiation, integer division, and multiplication can be recovered:

$$\begin{array}{ll} x \dot{-} y = (2^{x+y} + x) \,\%\, (2^{x+y} + y) \,\%\, (2^{x+y} + x) & x^y = 2^{(xy+x+1)y} \,\%\, (2^{xy+x+1} \dot{-} x) \\ x \div y = (2(x + 1)(x \dot{-} (x \,\%\, y))) \,\%\, (2(x + 1)y \dot{-} 1) & xy = ((x + y)^2 \dot{-} x^2 \dot{-} y^2) \div 2 \end{array}$$

Equalities and inequalities between functions, as well as their boolean combinations (negations, conjunctions, and disjunctions), can be equivalently expressed in the form $H(n, m, \ldots) = 0$:

[2] Can we establish a bi-interpretability between Gödel's System `T` and `ERTT` + ε_0-induction?

$$
\begin{array}{lcl}
F(\ldots) \neq 0 & \Leftrightarrow & 1 \dot{-} F(\ldots) = 0 \\
(F(\ldots) = 0) \vee (G(\ldots) = 0) & \Leftrightarrow & F(\ldots) \cdot G(\ldots) = 0 \\
(F(\ldots) = 0) \wedge (G(\ldots) = 0) & \Leftrightarrow & F(\ldots) + G(\ldots) = 0 \\
F(\ldots) \leqslant G(\ldots) & \Leftrightarrow & F(\ldots) \dot{-} G(\ldots) = 0 \\
F(\ldots) = G(\ldots) & \Leftrightarrow & (F(\ldots) \dot{-} G(\ldots)) + (G(\ldots) \dot{-} C(\ldots)) = 0
\end{array}
$$

Hence, quite sophisticated propositions can be reduced to the form $F = 0$, where `F` is an elementary recursive function of zero or more variables. As we will discuss below, the consistency and effective completeness of arbitrary axiomatic theories can be expressed in this form. At the same time, propositions of this form are experimentally falsifiable, given enough computational resources. If $H(\ldots) = 0$ is not true, we are bound to find a counterexample by computing $H(\ldots)$ for increasing values of its arguments.

The language of elementary recursive functions of zero or more variables can be given as a finitary inductive type (quotient inductive, if the defining identities for operations are incorporated):

```
data ERAexpr
  Nil                `0`
  Pow(n : ERAexpr)   `₂n`
  Mod(n m : ERAexpr) `n % m`
  Add(n m : ERAexpr) `n + m`
  Var(idx : ℕ)

  AddZero(n : ERAexpr) : n + 0 = n
  ModZero(n : ERAexpr) : n % 0 = n
  PowSucc(n : ERAexpr) : ₂(n + ₂0) = ₂n + ₂n
  ModSucc(n m : ERAexpr) : n % (n + m) = n
  AddSucc(n m : ERAexpr) : n + (m + ₂0) = (n + m) + ₂0
  SuccMod(n m : ERAexpr) : (n + m) % m = n % m
```

Let us also define how to convert from natural numbers to `ERA` expressions, how to compute the number of variables in an expression, and how to substitute an expression for a variable:

```
def toERAexpr : ℕ → ERAexpr
  0 ↦ 0
  n⁺ ↦ (toERAexpr n) + 1

def nVariables : ERAexpr → ℕ
  Nil ↦ 0
  Var(i) ↦ i + 1
  Pow(n) ↦ nVariables n
  Add(n, m) ↦ (nVariables n) max (nVariables m)
  Mod(n, m) ↦ (nVariables n) max (nVariables m)

def subst(i : ℕ, x : ERAexpr) : ERAexpr → ERAexpr
  Nil ↦ Nil
  Var(i) ↦ x
  Pow(n) ↦ Pow(n ▸ subst(i, x))
  Add(n, m) ↦ Add(n ▸ subst(i, x), m ▸ subst(i, x))
  Mod(n, m) ↦ Mod(n ▸ subst(i, x), m ▸ subst(i, x))
```

The fact that closed expressions (expressions with zero variables) can be evaluated can be stated type-theoretically as the function of the following signature:

```
def eval(e : ERAexpr, nVariables e = 0) : {n : ℕ | e = toERAexpr n}
```

That is, it evaluates a given expression with zero variables into a natural number, which is equal to the original expression along the identities defining basic elementary recursive operations.

With substitution we can also define non-falsifiability of `F(n) = 0` type-theoretically:

```
def isZero(F : ERAexpr, nVariables F = 1) = ∀(n : ℕ) F ▸subst(0, toERAexpr n) = 0
```

`ERAexpr` with defining identities for basic elementary recursive functions is not just a language but an equational theory, an axiomatic theory with axioms given by the identities and the usual inference rules for equality:

$$\frac{}{\texttt{A = A}}\text{refl} \quad \frac{\texttt{A = B}}{\texttt{B = A}}\text{sym} \quad \frac{\texttt{A = B} \quad \texttt{B = C}}{\texttt{A = C}}\text{trans} \quad \frac{\texttt{A = B} \quad \texttt{F(…,n,…) = G(…,n,…)}}{\texttt{F(…,A,…) = G(…,B,…)}}\text{subst}$$

This theory is too weak to prove any meta-theoretic results as it lacks any form of induction. Elementary recursive arithmetic is an extension of this theory by an inference rule for induction:

$$\frac{\texttt{F(0) = G(0)} \quad \texttt{F(n⁺) = H(n, F(n))} \quad \texttt{G(n⁺) = H(n, G(n))}}{\texttt{F(n) = G(n)}}\text{ind}$$

This rule can also be incorporated directly into the definition of `ERAexpr`, making it a finitary quotient inductive-*recursive* type, because the function `subst` is now used in the type definition:

```
Ind(i j : ℕ, F G H : ERAexpr,
    succF : F ▸ subst(i, (Var i) + ₂0) = H ▸ subst(0, (Var i)) ▸ subst(1, F),
    succG : G ▸ subst(j, (Var j) + ₂0) = H ▸ subst(0, (Var j)) ▸ subst(1, G),
    base : F ▸ subst(i, Nil) = G ▸ subst(j, Nil)) : F = G
```

The identity types $\mathrm{Id}_{\texttt{ERAexpr}}\ 0\ F$ (which are themselves finitary inductive types) are the types of `ERA`-proofs of `F = 0`, so we obtain the proof language for `ERA` for free. As one can observe, proofs within elementary recursive arithmetic have an extremely simple structure, and there are no inference rules nor axioms allowing proofs by contradiction, no possibility to refer to infinite or self-referential objects of any kind.

The type `ERAexpr : FinitaryType` and all of the above functions are definable inside `ERTT`. In `PRTT` we should be able to prove surjectivity of the above `eval` function thus proving consistency of `ERA`.

Following the "Type Theory Should Eat Itself" by J. Chapman et al,, we should be able to encode the syntax of `ERCC` within `ERTT`, and use it to define an intepretation of `ERCC` in `ERA`. It should be possible to prove not just soundness of the interpretation, but also the conservativity of `ERCC` over `ERA` within `ERTT`.

It should also be quite instructive to pinpoint exact proof-theoretic strength of `ERCC` and `ERCC` in `ERTT`, by proving `α`-induction for all ordinals below `ω³` and prove the consistency of `ERCC` relative to `ω³`-induction.

4 Finitary constructruction calculus `FinCC`

We can go even below elementary recursive. Recently, F. Pakhomov devised a weak set theory $\texttt{H}_<$ that proves its own consistency. It evades the curse of second Gödel's inconsistency theorem by being a locally finitely satisfiable theory, and thus having the lowest possible infinite proof-theoretic strength `ω`. To obtain its type-theoretic counterpart we have restrict codomains of all unbounded functions[3] to finite types, while allowing polymorphic definitions. Instead of defining an unbounded function, we define an infinite family of its bounded approximations. Effectively, we define unbounded functions relative to the condition "as long as there are enough natural numbers" or "as long as there is enough memory space to compute".

Application of a constructor of an inductive type `J` increases the amount of required "memory space", so we have to ensure we have enough. So we require an initial cut `r` of `J`, where the

[3]Codomoains without eliminators (such as `Type` and `Prop`) and mere propositions `P : Prop` are also fine.

result would lie, as a parameter, and define the constructor only for elements of a subcut `n` of `r` to ensure they are small enough. The signature of `succ` (in fact, of any constructor) looks like thus:

```
succ : ∀<r : ℕ, n : r> n → r
```

To accomodate iterated application of constructors, we need additional machinery. Crusially, elementary recursive functions use iteration in a very controlled manner: they have finite iteration depth, and each the number of executions of each loop is bounded by the size of the original arguments. This way, any elementary recursive function can be implemented as a polymorphic finite-codomain function by relativizing to a superexponential cut. This is where set-theoretic interpretation of terms becomes very important. In the first section we mentioned that every inhabitant `t : Q` of a finitary inductive type obtains a natural structure of a hereditarily finite multiset. Let `V̄(t)` denote the von Neumann-closure of this hereditary multiset, and let `J @ t` denote the finite subtype of `J` only consists of terms that are embeddable as multisets into `V̄(t)`. Then the cut `t` (the subtype of `Q` containing subexpressions of `t`) is superexponentially larger than any of the cuts `x : J @ t`.

Now consider an elementary recursive function `f(n : J) → R`. If it only uses `n` for induction, and induction steps only once apply a constructor, its signature can be written as follows in finitary construction calculus:

```
f : ∀<r : R, m : J @ r> m → r
```

If induction employs `n` constructors, the signature should be as follows:

```
f : ∀<r : R, m₁ : J @ r, m₂ : J @ m₁, ...> mₙ→ r
```

Since every elementary recursive function only applies induction a finite number of times and each instance contains a finite number of constructiors, any elementary recursive function can be given a signature inside `FinCC`. We still retain enough expressivity to state metamathematical results, construct translations/interpretations/models while proving their soundness by construction. But we cannot prove unrestricted induction for any infinite ordinals (as desired). It seems it also should still be possible to prove strong normalization results relative to induction along ordinal notations.

As we mentioned in the beginning, `FinCC` constructors of inductive types and, in fact, all unbounded functions are defined by infinite families of finitary definitions. If we only take a finite number of instances from these infinite families, the resulting theory admits a finite model: this property is knonw as being locally finitely satisfiable. Since any proof `prf : P : Prop` in `FinCC` only makes use of a finite number of definitions, it can be checked in a finite model of `FinCC`, namely in the superexponential cut of hereditarily finite multisets $\mathtt{V}_{\mathtt{FinitaryType}}$.

It should be possible to define the syntax for `FinCC` proofs as an inductive type `PrfTree` together with map `conclusion : PrfTree → Prop`, and procedure that checks `p : PrfTree` inside the model $\mathtt{V}_{\mathtt{FinitaryType}}$ `@ p`, yielding an internal consistency result

```
∀(p : PrfTree) Conclusion p
```

Hilbert's dream of a finitisitc theory recognizing its own consistency while being capable of metatheoretic reasoning about much more intricate theories, might be attainable after all.