RZ: a Tool for Bringing Constructive and Computable Mathematics Closer to Practice

Andrej Bauer   Chris Stone

Department of Mathematics and Physics
University of Ljubljana, Slovenia

Computer Science Department
Harvey Mudd College, USA

CiE 2007, Siena, June 2007

Theory and practice

- The theory of constructive & computable mathematics:
  - Structures from analysis and topology are studied.
  - *Informal descriptions* of algorithms via Turing machines.
  - Deals mostly with: “What *can* be computed?”
  - Efficiency of computation is desired.
- Practice of computing:
  - Emphasis on discrete mathematics.
  - *Implementations* of practical data structures and algorithms.
  - Deals with: “How *fast* can we compute?”
  - Speed is essential.

Can we bring constructive math closer to practice?

- Sacrificing performance for correctness is unacceptable.
  - Currently programs extracted from formal proofs are inefficient.
  - Programmers should be free to implement efficient code.
  - Provide support for proving correctness of implementation.
- It is tricky to correctly implement structures from analysis and topology.
  - We should link mathematical models with practical programming.
  - Give programmers tools that automate tasks.

Our contribution

- A theory of representations based on Objective Caml.
  - We replaced Turing machines (type I and II) with a real-world programming language.
  - Representations can *actually* be implemented.
  - Other programming languages could be used.
- But we do not work with representations directly.
  - Cumbersome and generally too low a level of abstraction to do mathematics.
  - How do we know which representation of a given set is the right one?
- Instead, we use representations as a model in which to interpret constructive mathematics.
  - Use Kleene's realizability interpretation adapted to OCaml.
  - The translation of a constructive theory is a *specification* describing how to implement it in OCaml.
- Most importantly, we built a tool RZ which *automatically* translates constructive logic to representations.

Representations

- Representations are a successful idea in computable mathematics:
  - numbered sets,
  - Type Two Effectivity representations,
  - domain-theoretic representations,
  - equilogical spaces.
- Phrased in various forms:
  - partial surjections,
  - partial equivalence relations,
  - modest sets,
  - assemblies,
  - multi-valued partial surjections,
  - realizability relations.
- Can be described to programmers without much trouble.

Representations in Objective Caml

- A representation $\delta : \mathtt{t} \to X$ consists of:
  - represented set X
  - representing datatype $\mathtt{t}$
  - partial surjection $\delta : \mathtt{t} \to X$
- Define the partial equivalence relation (per) $\approx$ on $\mathtt{t}$ by

$$u \approx v \iff u, v \in \mathrm{dom}(\delta) \land \delta(u) = \delta(v) \; .$$

- We may recover $\delta : \mathtt{t} \to X$ from $(\mathtt{t}, \approx)$ up to isomorphism:

$$\|\mathtt{t}\| = \{u \in \mathtt{t} \mid u \approx u\}$$
$$X \cong \|\mathtt{t}\|/{\approx}, \quad \mathrm{dom}(\delta) = \|\mathtt{t}\|, \quad \delta(u) = [u]_{\approx}$$

- Note: δ and $\approx$ are *not* required to be computable, they live "outside" the programming language.

Constructions of representations

Representations, together with a suitable notion of morphisms, form a rich category with many constructions:

- products $A \times B$ and disjoint sums $A + B$,
- function spaces $A \to B$,
- dependent sums $\Sigma_{i \in A} B(i)$ and products $\Pi_{i \in A} B(i)$,
- subsets $\{x : A \mid \phi(x)\}$,
- quotients A/ρ,
- but *no* powersets.

This is a convenient "toolbox" for constructive mathematics.

Realizability interpretation of logic

- A formalization of Brouwer-Heyting-Kolmogorov interpretation of intuitionistic logic.
- Validity of a proposition ϕ is witnessed by a *realizer*:

$$r \Vdash \phi \qquad \text{“}r\text{ is computational witness of }\phi\text{”}$$

- Note: r could be any OCaml value, need not correspond to a proof under the Curry-Howard correspondence.
- The type of r and $\Vdash$ are defined inductively on the structure of ϕ, e.g.:

$$\begin{array}{lll} \langle r_1, r_2 \rangle \Vdash \phi_1 \wedge \phi_2 & \text{iff} & r_1 \Vdash \phi_1 \text{ and } r_2 \Vdash \phi_2 \\ r \Vdash \phi \implies \psi & \text{iff} & \text{whenever } s \Vdash \phi \text{ then } r(s) \Vdash \psi \\ \dots & & \dots \end{array}$$

RZ

- Input: one or more theories
- Output: OCaml module type specifications
- Translation has several phases:
  1. Type-checking: does the input make sense?
  2. Translation via realizability interpretation
  3. Thinning: remove computationally irrelevant realizers
  4. Optimization: perform further simplifications to output
  5. Phase splitting (will not explain here, read the paper)

Input

A theory consists of declarations, definitions, and axioms.

```
Definition Ab :=
thy
  Parameter t : Set.
  Parameter zero : t.
  Parameter neg : t → t.
  Parameter add : t ⋆ t → t.
  Definition sub (u : t) (v : t) := add(u, neg v).
  Axiom zero_neutral: ∀ u : t, add(zero,u) = zero.
  Axiom neg_inverse: ∀ u : t, add(u,neg u) = zero.
  Axiom add_assoc:
   ∀ u v w : t, add(add(u,v),w) = add(u,add(v,w)).
  Axiom abelian: ∀ u v : t, add(u,v) = add(v,u).
end.
```

Theories can be *parametrized*, e.g., the theory of a vector space parametrized by a field, `VectorSpace(F:Field)`.

Translation and output

- Consider the input:

```
Axiom lpo : ∀ f : nat → nat,
  [`zero: ∀ n : nat, f n = zero] ∨
  [`nonzero: ¬ (∀ n : nat, f n = zero)].
```

- In the output we get a *value declaration* and an *assertion*:

```
val lpo : (nat → nat) → [`zero | `nonzero]
(**  assertion lpo :
  ∀ (f:‖nat → nat‖),
    (match lpo f with
       `zero ⇒ ∀ (n:‖nat‖), f n ≈nat zero
     | `nonzero ⇒ ¬ (∀ (n:‖nat‖), f n ≈nat zero))
*)
```

- The value `lpo` is the computational content of the axiom.
- An implementation of `lpo` must satisfy the assertion.
- Assertion is free of computational content, thus its constructive and classical readings agree.

Example: "All functions are continuous"

- Input:

```
Axiom modulus:
∀ f : (nat → nat) → nat, ∀ a : nat → nat,
  ∃ k : nat, ∀ b : nat → nat,
    (∀ m : nat, m ≤ k → a m = b m) → f a = f b.
```

- RZ output:

```
val modulus : ((nat → nat) → nat) → (nat → nat) → nat
(**  Assertion modulus =
  ∀ (f:‖(nat → nat) → nat‖, a:‖nat → nat‖),
    let p = modulus f a in p : ‖nat‖ ∧
    (∀ (b:‖nat → nat‖),
      (∀ (m:‖nat‖), m ≤ p → a m ≈nat b m) →
      f a ≈nat f b) *)
```

- Implementation:

```
let modulus f a =
  let p = ref 0 in
  let a' n = (p := max !p n; a n) in
    ignore (f a') ; !p
```

Remarks

- We have implemented real numbers using RZ:
  - see Bauer & Kavkler at CCA 2007.
- We would like to implement more advanced structures:
  - manifolds, Hilbert spaces, analytic functions, …
  - we expect these to be painfully slow at first.
- Even if you do not want to implement anything, you can use RZ to *automatically* compute representations from constructive definitions.
- It would be interesting to connect RZ with a tool that allows formal verification of correctness, such as Coq.