Mathematically Structured but not Necessarily Functional Programming

Andrej Bauer

Department of Mathematics and Physics
University of Ljubljana, Slovenia

Mathematically Structured Functional Programming
Reykjavik, July 2008
Ways of Mathematically Structured Programming

- Use math to develop new programming constructs (monads).
- Use math to reason and construct programs (Coq).
- Programming by proving theorems (propositions as types).
- Proving theorems by programing (types as propositions).
Outline

- Programming = Proving (propositions as types)
- Programming = Proving (realizability)
- RZ – specifications via realizability
- Examples of non-functional realizers in constructive mathematics
Programming by proving

- The Curry-Howard correspondence:
  \[
  \text{Type} = \text{Prop} = \text{Set} \\
  \text{program} = \text{proof} = \text{element}
  \]

- Programming by proving theorems:
  
  "Constructive proofs of mathematically meaningful theorems give useful programs."
Example: Fundamental Theorem of Algebra

- “Every non-constant polynomial has a complex root.”
- First-order logic:

\[ \forall p \in \mathbb{Q}[x]. \ 0 < \text{deg}(p) \implies \exists z \in \mathbb{C}. \ p(z) = 0. \]

- Type theory:

\[ \prod_{p: \text{poly}} \text{less}(0, \text{deg}(p)) \to \sum_{z: \text{complex}} \text{eq}(p(z), 0). \]

- Must also define \text{poly}, \text{less}, \text{complex}, and \text{eq}.
- Can we get rid of \text{less} and \text{eq}?
- Can we get rid of dependent types and have just

\[ \text{poly} \to \text{complex} ? \]
Programming by proving a la Coq

- Distinguish between computational and non-computational types:
  
  \[ \text{Set} : \text{the sort of computational types} \]
  
  \[ \text{Prop} : \text{the sort of non-computational types} \]

- We also need \textit{setoids}, which are (computational) types with (non-computational) equivalence relations.

- In the previous example:
  
  - Non-computational: \text{less}, \text{eq}.
  - \text{Setoids}: \text{poly}, \text{complex}.

- Coq’s extraction mechanism gives an Ocaml or Haskell program of type \text{poly} \rightarrow \text{complex}.
Does it actually work?

- Programmers want to write programs, not proofs.
- And often it really is easier to just write a program.
- The most efficient proof may not correspond to the most efficient program.
- When we use complex tactics, we may lose control of what the extracted program does.
- Proofs give purely functional code. What if we want to use computational effects (store, exceptions, non-termination)?
What really happens

- Write programs directly, not as proofs.
- Then prove that the programs are correct.
- Coq’s PROGRAM extension does this.
- By adapting the type theory and the extraction mechanism, we can even handle non-functional programs.

The connection to constructive math is almost lost.
Programming by proving (a la realizability)

- Pick a reasonable programming language.
- Proofs $\subseteq$ Programs.
- Programs realize propositions.
- To each proposition $\phi$ we assign a (simple) type of realizers $|\phi|$.
- We define a realizability predicate on values of $|\phi|$: $p \Vdash \phi$ “$p$ realizers $\phi$.”

This is necessary because not every value in $|\phi|$ is a valid realizer.
Types of realizer

\[ |T| = \text{unit} \]
\[ |⊥| = \text{unit} \]
\[ |e_1 =_A e_2| = \text{unit} \]
\[ |φ_1 \land φ_2| = |φ_1| \times |φ_2| \]
\[ |φ_1 \lor φ_2| = |φ_1| + |φ_2| \]
\[ |φ_1 \implies φ_2| = |φ_1| \rightarrow |φ_2| \]
\[ |∀x ∈ A. φ| = |A| \rightarrow |φ| \]
\[ |∃x ∈ A. φ| = |A| \times |φ| \]

Propositions built only from \(T, ⊥, =, \land, \rightarrow\) have trivial realizers.
Realizability predicate

$() \vdash \top$

$() \vdash e_1 =_A e_2 \quad \text{iff} \quad t_1 \simeq_A t_2$

$(p_1, p_2) \vdash \phi_1 \land \phi_2 \quad \text{iff} \quad p_1 \vdash \phi_1 \quad \text{and} \quad p_2 \vdash \phi_2$

$\text{inl}(p) \vdash \phi_1 \lor \phi_2 \quad \text{iff} \quad p \vdash \phi_1\quad $ (p, q) \vdash \exists x \in A. \phi(x) \quad \text{iff} \quad \text{for some } u, q \vdash_A u \quad \text{and} \quad p \vdash \phi(u)$

$p \vdash \phi_1 \implies \phi_2 \quad \text{iff} \quad \text{if } q \vdash \phi_1 \text{ then } p q \downarrow \text{ and } p q \vdash \phi_2$

$(p, q) \vdash \forall x \in A. \phi(x) \quad \text{iff} \quad \text{if } q \vdash_A u \text{ then } p q \downarrow \text{ and } p q \vdash \phi(u)$
Setoids in realizability

- In realizability setoids are types equipped with *partial* equivalence relations (symmetric, transitive).
- This is necessary because not every value realizes an element.
- Even when the programming language is simply typed, we can interpret dependent setoid types.
A tool written by Chris Stone and me. It uses realizability to translate mathematical theories to program specifications.

Input: mathematical theories
- first-order logic
- rich set constructions, including dependent types
- support for parameterized theories, e.g., the theory of a vector space parameterized by a field.

Output: program specifications
- Ocaml signatures
- Assertions about programs

Automatically eliminates non-computational realizers.
Test case: Era

- A package for exact real numbers.
- Written by Iztok Kavkler and me.
- What we did:
  - wrote down theories of $\omega$-cpos, the interval domain and real numbers,
  - translated them to specifications with RZ,
  - implemented the specification efficiently.
- Conclusion: it works, but we have no tool to prove that our programs satisfy the assertions.
- Plan: extend RZ so that it translates to Coq using the PROGRAM extension.
Non-functional realizers

- There are constructive reasoning principles which cannot be proved in pure intuitionistic logic.
- They cannot be realized in pure type theory or pure Haskell.
- They are realized by non-functionals programs.
- Such principles express the mathematical meaning of non-functional programs.
Markov Principle

- “A sequence of 0’s and 1’s whose terms are not all 0 contains a 1.”
- “A program which does not run forever terminates.”
- Provable in classical logic.
- Cannot be proved in intuitionistic logic.
- $\forall a : \{0, 1\}^\mathbb{N}. (\neg\forall n : \mathbb{N}. a(n) = 0) \implies \exists n : \mathbb{N}. a(n) = 1.$
- RZ tells us that the realizer has type $(\text{nat} \to \text{bool}) \to \text{nat}.$

- Realized by unbounded search:

```
let mp a =
    let n = ref 0 in
    while not (a !n) do n := !n + 1 done ; !n
```
Brouwer’s Continuity Principle

- “Every map is continuous.”
- “Every map $f : \mathbb{N}^\mathbb{N} \to \mathbb{N}$ is continuous.”
- In other words, $f(a)$ depends only on a finite prefix of $a(0), a(1), a(2), \ldots$.
- Incompatible with classical logic.
- Cannot be proved in intuitionistic logic.
- As a formula:

  \[ \forall f \in \mathbb{N}^{\mathbb{N}^\mathbb{N}}. \forall a \in \mathbb{N}^{\mathbb{N}}. \exists n \in \mathbb{N}. \forall b \in \mathbb{N}^{\mathbb{N}}. \quad ((\forall k \leq n. a(k) = b(k)) \implies f(a) = f(b)). \]

- Realizers of type

  \[ ((\text{nat} \to \text{nat}) \to \text{nat}) \to (\text{nat} \to \text{nat}) \to \text{nat} \]
Continuity principle with store

- How can we discover how many terms of \(a(0), a(1), \ldots\) are used by \(f\)?
- Feed \(f\) a sequence which is just like \(a\), except that it also stores the largest argument at which \(f\) evaluated it.
- The code:

```ocaml
let cont f a =
  let k = ref 0 in
  let b n = (k := max !k n; a n) in
  f b ; !k
```
Continuity principle with exceptions

- Similar idea: throw an exception if \( f \) looks past a threshold, and keep increasing the threshold until no exception is raise.

- The code

```ocaml
exception Abort
let cont f a =
  let rec search k =
    try
      let b n =
        if n < k then a n else raise Abort
      in
      f b ; k
    with Abort -> search (k+1)
  in
  search 0
```
Can we prove these realizers work?

- Store: presumably yes, using separation logic.
- But with global store it does *not* work:
  ```haskell
  let k = ref 0
  let cont f a =
    let b n = (k := max !k n; a n) in
    f b ; !k
  ```
- This version is foiled by
  ```haskell
  let f a =
    let m = a 42 in k := 0 ; m
  ```
- Note: Haskell’s *State* monad is global store.
Realizer with exceptions does not work!

- The realizer using exceptions does not work.
- Foiled by
  
  ```ml
  let f a =
    try a 42 with Abort -> 23
  ```

- Even if `Abort` is declared locally, we can still catch all exceptions in ML:
  
  ```ml
  let f a =
    try a 42 with _ -> 23
  ```

- Haskell also has global exceptions.
Conclusion

- Realizability is a useful alternative to propositions as types.
- *We can* keep the connection between constructive math and programming tight, without sacrificing either mathematical elegance or efficiency of programs.
- Constructive reasoning principles are a mathematical abstraction of non-functional programming features.
- We need to study non-functional features more carefully.