Mathematically Structured but not Necessarily Functional Programming (Extended Abstract)

Andrej Bauer[1]

Faculty of Mathematics and Physics
University of Ljubljana
Ljubljana, Slovenia

Abstract

Realizability is an interpretation of intuitionistic logic which subsumes the Curry-Howard interpretation of propositions as types, because it allows the realizers to use computational effects such as non-termination, store and exceptions. Therefore, we can use realizability as a framework for program development and extraction which allows any style of programming, not just the purely functional one that is supported by the Curry-Howard correspondence. In joint work with Christopher A. Stone we developed RZ, a tool which uses realizability to translate specifications written in constructive logic into interface code annotated with logical assertions. RZ does not extract code from proofs, but allows any implementation method, from handwritten code to code extracted from proofs by other tools. In our experience, RZ is useful for specification of non-trivial theories. While the use of computational effects does improve efficiency it also makes it difficult to reason about programs and prove their correctness. We demonstrate this fact by considering non-purely functional realizers for a Brouwerian continuity principle.

Keywords: Realizability, constructive mathematics, specifications, functional programming, computational effects.

Realizability, a notion introduced by Kleene [5], is a mathematical formalization of the Brouwer-Heyting-Kolmogorov interpretation of intuitionistic logic. Under the realizability interpretation, validity of a proposition is witnessed by a program, or a *realizer*, which can be thought of as the computational content of the proposition. For example, a $\forall x.\, \exists y.\, \phi(x, y)$ statement is valid if there is a (not necessarily extensional) program computing from x a y and a realizer for $\phi(x, y)$. Realizability subsumes the Curry-Howard correspondence which interprets proposition as types and proofs as programs, because a realizer need not correspond to a proof—it may be non-terminating, or use computational effects such as store and exceptions.

We can use realizability as a mathematical framework for program extraction and development, just as the Curry-Howard correspondence is used for the same purpose in systems such as Coq [3] and Agda [6]. Since realizers need not correspond to proofs, they are generally obtained as a combination of handwritten code and

[1] Email: Andrej.Bauer@andrej.com

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

code extracted from constructive proofs. This allows us to realize statements which are not provable in intuitionistic logic. Even when a purely functional realizer could be extracted from a proof, we might prefer an impure handwritten one because it is more efficient, or because it is easier to write the code than the proof. In fact, an important advantage of realizability is the fact that it allows programmers to implement specifications in any way they see fit. [2]

In joint work with Chris Stone we developed *RZ* [2], a tool which employs the realizability interpretation to translate specifications in constructive logic into annotated interface code in Objective Caml [7]. The system supports a rich input language allowing descriptions of complex mathematical structures. Currently, RZ does not extract code from proofs, but allows any implementation method, from handwritten code to code extracted from proofs by other tools. Recently, we have also been experimenting with translation into Coq using Matthieu Sozeau's PROGRAM extension [8]. In a related project with Iztok Kavkler [1] we demonstrated that RZ can be used to develop non-trivial specifications. We wrote an RZ axiomatization of real numbers and the interval domain. We then implemented by hand the specification produced by RZ, which gave us a remarkably efficient library *Era* for computing with exact real numbers. Most of Era is written in a purely functional style, and we could have easily extracted the realizers from constructive proofs. However, the ability to use non-functional features such as caching and in-place updates, improves performance and eases implementation.

We demonstrate the use of non-purely functional realizers by looking at the statement

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

This is a continuity principle saying that every function from Baire space $\mathbb{N}^{\mathbb{N}}$ to $\mathbb{N}$ is continuous, where $\mathbb{N}$ is equipped with the discrete and $\mathbb{N}^{\mathbb{N}}$ with the product topology. It has important consequences for the theory of metric spaces in Brouwerian intuitionistic mathematics. We cannot hope to prove the statement in pure intuitionistic logic because such a proof would also be valid classically, but the statement is incompatible with classical logic. [3] By feeding the statement to RZ we find out that it is realized by a realizer

$$\mathtt{r} : ((\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat}) \to (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat}$$

that takes as input realizers $\mathtt{f} : (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat}$ and $\mathtt{a} : \mathtt{nat} \to \mathtt{nat}$, which realize the corresponding functions f and a, and returns a realizer $\mathtt{n} = \mathtt{r}\,\mathtt{f}\,\mathtt{a}$ of type $\mathtt{nat}$ such that $\mathtt{f}$ depends on at most the first $\mathtt{n}$ terms of the sequence $\mathtt{a}(0), \mathtt{a}(1), \mathtt{a}(2), \ldots$ Such a realizer is known as *modulus of continuity*. We cannot expect to construct $\mathtt{r}$ in a purely functional language, [4] although there are a number of ways of getting it

[2] Whether the programmers also go to the trouble of proving correctness of handwritten code is another matter.

[3] Consider f defined by $f(a) = 1$ if $\exists n \in \mathbb{N}. a(n) = 1$, and $f(a) = 0$ otherwise. An intuitionistic proof applied to f would essentially give a Halting Oracle.

[4] This is so because there are domain-theoretic models which interpret purely functional code and invalidate the continuity principle.

with the aid of computational effects. For example, by using store we can implement `r` as (written in Objective Caml notation)

```
let r f a =
  let k = ref 0 in
  let b n = (k := max !k n; a n) in
    f b ; !k
```

Given `f` and `a`, `r` evaluates `f` at the argument `b` which behaves just like `a`, except that it stores in location `k` the largest argument at which it was evaluated. When `f` returns a value, `r` discards it and returns the number stored in `k`.

We could use exceptions instead of store:

```
exception Abort
let r 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
```

Now the argument `b` acts like `a` for arguments below the threshold value `k`, and raises an exception otherwise. The program `r` keeps increasing `k` as long as exceptions are being raised. Eventually it finds a threshold which does not raise an exception and returns it. We leave it to the readers to construct `r` which uses continuations.

It is not easy to prove that the realizer `r` really does what it is supposed to. In fact, the first version, using store, only works because `k` is local and thus inaccessible to `f`. Had we used a globally accessible location `k`, the following `f` would foil `r` by resetting `k` to 0:

```
let f a =
  let m = a 42 in k := 0 ; m
```

It is somewhat unclear whether `f` realizes an element of $\mathbb{N}^{\mathbb{N}^{\mathbb{N}}}$. We remark that Haskell `State` monad [4] implements global store. It is an interesting question how to get `r` in Haskell using only monads that are definable in pure Haskell. [5]

The situation is even worse with the version of `r` that uses exceptions. It simply does not work because `f` could intercept the exception:

```
let f a =
  try a 42 with Abort -> 23
```

Note that `f` is allowed to do this because it only has to work properly on arguments which do not raise exceptions. To mend the situation, we would need a local exception `Abort` which cannot be intercepted by `f`. While exceptions in Objective Caml can be declared locally, they can also be intercepted globally. One is tempted to either ask the designers of the language for truly local exceptions, or to formulate

[5] We are asking for a monad `T` such that `r` can be written in the Kleisli category for `T`, i.e., a function accepting type α and returning type β has type $\alpha \to \mathtt{T}\,\beta$.

an additional invariant that `f` should satisfy, such as "does not intercept exception `Abort`". Neither alternative is appealing.

These considerations reflect the well known wisdom that it not easy to reason about computational effects. However, we feel that realizability still provides a convenient mathematical framework for development and extraction of programs and data structures, especially since it does not impose a particular programming style.

References

[1] Bauer, A. and I. Kavkler, *Implementing real numbers with RZ*, Electronic Notes in Theoretical Computer Science **202** (2008), pp. 365–384.

[2] Bauer, A. and C. A. Stone, *RZ: A tool for bringing constructive and computable mathematics closer to programming practice*, in: S. B. Cooper, B. Löwe and A. Sorbi, editors, *Computation and Logic in the Real World*, Lecture Notes in Computer Science **4497**, Springer Berlin/Heidelberg, 2007 pp. 28–42.

[3] Bertot, Y. and P. Castéran, "Interactive Theorem Proving and Program Development," Springer, 2004.

[4] Haskell, *http://www.haskell.org/*.

[5] Kleene, S., *On the interpretation of intuitionistic number theory*, Journal of Symbolic Logic **10** (1945), pp. 109–124.

[6] Norell, U., "Towards a practical programming language based on dependent type theory," Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden (2007).

[7] Objective Caml, *http://www.ocaml.org/*.

[8] Sozeau, M., *Program-ing finger trees in coq*, in: *ICFP'07: Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming* (2007), pp. 13–24.