Specifications via Realizability

Andrej Bauer [1]

Department of Mathematics and Physics
University of Ljubljana
Ljubljana, Slovenia

Christopher A. Stone [2]

Computer Science Department
Harvey Mudd College
Claremont, CA, USA

Abstract

We present a system, called RZ, for automatic generation of program specifications from mathematical theories. We translate mathematical theories to specifications by computing their realizability interpretations in the ML language augmented with assertions (as comments). While the system is best suited for descriptions of those data structures that can be easily described in mathematical language (e.g., finitely presented groups, real arithmetic, graphs, etc.), it also elucidates the relationship between data structures and constructive mathematics.

Key words: Realizability, Constructive Logic, ML.

1 Introduction

Kleene [6] introduced realizability as a model of intuitionistic arithmetic based on partial computable functions. The idea has since been studied and generalized by various authors [11,4,5,13]. Building on the idea of *typed realizability* by Longley [8], we have constructed a tool RZ to translate mathematical theories into specifications for code, explaining what is necessary in order to believe that we have a correct implementation of the mathematical theory.

As the realizability interpretation validates the laws of *intuitionistic* logic, our input theories are intuitionistic or constructive. Thus, RZ extracts the computational meaning of a constructive theory and expresses it as a programming specification.

[1] Email: Andrej.Bauer@andrej.com
[2] Email: stone@cs.hmc.edu

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

We emphasize that RZ does *not* extract programs from proofs—in fact, there is no way to write a proof in our system. We just determine what the programs are supposed to do, i.e., we provide specifications for them. We leave it to the programmer, or to another tool, to construct the programs as he or she sees fit. This leaves the programmer completely free to write efficient programs that need not correspond directly to a formal proof.

The original aim of RZ was to aid development of data structures for computable mathematics. If one sets out to actually compute realizability interpretations of theories of constructive mathematics, one quickly wishes for an automated way of doing it. With a tool like RZ it is much easier to experiment and try out variations of a theory until a suitable specification is obtained. It also appears that RZ can be used to explain and teach constructive mathematics to programmers, who are typically trained in classical mathematics; it translates constructive statements into easily understood requirements about programs (expressed in classical logic).

Our implementation of RZ produces interfaces in Objective Caml [7] but could easily be adopted to other similar typed languages. The essential features we require of the target language are product, function, and sum types, as well as support for module interfaces.

The paper is organized as follows. Section 2 contains a brief overview of realizability. In Section 3 we describe theories and signatures, which are the inputs and the outputs of RZ, respectively. In Section 4 we discuss various point of implementation. In Section 5 we show typical examples and conclude with Section 6.

2 Realizability

We briefly motivate the main idea of (typed) realizability. When we represent a set of mathematical objects S in a programming language $\mathcal{P}$ there are two natural steps to take: first choose an *underlying type* $|S|$ of representing values, and second specify how the values of type $|S|$ represent, or *realize*, elements of the set S. For example, consider how we might represent the set D of simple finite directed graphs (whose vertices are labeled by integers). As the underlying datatype we might choose $|D| = \texttt{int list} * (\texttt{int} * \texttt{int})\texttt{list}$, and represent a graph $G \in D$ as a pair of lists (v, e) where $v = [x_1; \ldots; x_n]$ is the list of vertices and $e = [e_1; \ldots; e_m]$ is the list of edges. Formally, we write

$$(v, e) \Vdash_D G$$

and read it as "(v, e) realizes $G \in D$". Observe that each graph is realized by at least one pair of lists, and that no pair of lists represents more than one graph. (As commonly occurs, most graphs are represented by many different pairs of lists.) This leads us to the definition given below. We shall abuse notation slightly and write $t \in |S|$ to mean that t is a closed expression of type $|S|$.

$\lvert\top\rvert = \mathtt{unit}$	$() \Vdash \top$		
$\lvert\bot\rvert = \mathtt{unit}$			
$\lvert x =_S y\rvert = \mathtt{unit}$	$() \Vdash x =_S y$	iff	$x \approx_S y$
$\lvert\phi \wedge \psi\rvert = \lvert\phi\rvert * \lvert\psi\rvert$	$(t_1, t_2) \Vdash \phi \wedge \psi$	iff	$t_1 \Vdash \phi$ and $t_2 \Vdash \psi$
$\lvert\phi \implies \psi\rvert = \lvert\phi\rvert \to \lvert\psi\rvert$	$t \Vdash \phi \implies \psi$	iff	for all $u \in \lvert\phi\rvert$, if $u \Vdash \phi$ then $t\,u \Vdash \psi$
$\lvert\phi \vee \psi\rvert = \lvert\phi\rvert + \lvert\psi\rvert$	$\mathtt{inl}\,t \Vdash \phi \vee \psi$	iff	$t \Vdash \phi$
	$\mathtt{inr}\,t \Vdash \phi \vee \psi$	iff	$t \Vdash \psi$
$\lvert\forall x \in A.\,\phi(x)\rvert = \lvert A\rvert \to \lvert\phi\rvert$	$t \Vdash \forall x \in A.\phi(x)$	iff	for all $u \in \lvert A\rvert$, if $u \Vdash_A x$ then $t\,u \Vdash \phi(x)$
$\lvert\exists x \in A.\,\phi(x)\rvert = \lvert A\rvert \times \lvert\phi\rvert$	$(t_1, t_2) \Vdash \exists x \in A.\phi(x)$	iff	$t_1 \Vdash_A x$ and $t_2 \Vdash \phi(x)$

Fig. 1. Realizability interpretation of logic (outline)

Definition 2.1 A *modest set* [3] is a triple $(S, \lvert S\rvert, \Vdash_S)$ where S is a set, $\lvert S\rvert$ is a type and $\Vdash_S$ is a relation between expressions of type $\lvert S\rvert$ and elements of S, satisfying:

(i) For every $x \in S$ there is $t \in \lvert S\rvert$ such that $t \Vdash_S x$.

(ii) If $t \Vdash_S x$ and $t \Vdash_S y$ then $x = y$.

A *realized function* $f : (S, \lvert S\rvert, \Vdash_S) \to (T, \lvert T\rvert, \Vdash_T)$ between modest sets is a function $f : S \to T$ for which there exists $u \in \lvert S\rvert \to \lvert T\rvert$ such that

$$t \Vdash_S x \implies u\,t \Vdash_T f(x)\,.$$

We say that u *realizes* f.

The realizer u of a realized function f is more commonly known as an "implementation of f" or an "algorithm for computing f".

Modest sets and realized functions form a category of *modest sets* $\mathsf{Mod}(\mathcal{P})$. In realizability theory this is a well known category with good properties. It is regular and locally bi-cartesian closed, which allows us to interpret first-order logic and a rich type theory. Here we only outline the main ideas behind the realizability interpretation of logic. See e.g. [1] for details.

In the realizability interpretation of logic, each formula ϕ is assigned a set of *realizers* which can be thought of as computations that witness the validity of ϕ. The situation is somewhat similar (but not equivalent) to the propositions-as-types translation of logic into type theory, where the proofs of a proposition correspond to terms of the corresponding type. More precisely, to each formula ϕ we assign an underlying type $\lvert\phi\rvert$ of realizers. However, unlike in the propositions-as-types translation, not all terms of type $\lvert\phi\rvert$ are necessarily valid realizers for ϕ. We write $t \Vdash \phi$ when $t \in \lvert\phi\rvert$ is a realizer for ϕ.

[3] Modest sets were so named by Dana Scott. They are "modest" because their size cannot exceed the number of expressions of the underlying datatype.

The underlying types and the realizability relation $\Vdash$ are defined inductively on the structure of ϕ; an outline is shown in Figure 1. We say that a formula ϕ is *valid* in $\mathsf{Mod}(\mathcal{P})$ if it has at least one realizer.

We shall not dwell any further on the technicalities involving the category of modest sets, but rather proceed to a concrete description of our realizability translation. There is one technical point, though, which we first take care of. A modest set is a triple $(S, |S|, \Vdash_S)$ in which S is an arbitrary set. For an automated system it would be convenient if it did not have to refer to arbitrary sets but rather just to ingredients that are already present in the programming language, such as types and sets of expressions. Up to equivalence of categories, modest sets can be constructed as triples $(|S|, \|S\|, \approx_S)$ where $|S|$ is a type, $\|S\|$ is a subset of expressions of type $|S|$, called the *total values*,[4] and $\approx_S$ is an equivalence relation on $\|S\|$. The relationship between this representation of a modest set and the original one is as follows:

- $\|S\|$ is the set of those $t \in |S|$ that realize something, i.e., there is $x \in S$ such that $t \Vdash_S x$. These correspond to implementations that satisfy the representation invariant, e.g., graphs where the list of edges mentions only integers in the list of nodes, a subset of all values of type `int list` $*$ (`int` $*$ `int`) `list`.
- $t \approx_S u$ if t and u realize the same element, i.e., there is $x \in S$ such that $t \Vdash_S x$ and $u \Vdash_S x$. This relation equates alternate concrete representations of the same abstract value, e.g., equating two concrete graph representations differing only in the order of the nodes or the order of the edges.

The alternative view of a modest set $(|S|, \|S\|, \approx_S)$ only refers to objects and concepts from the programming language. It is better suited for our purposes.

Note that the equivalence relation on $\|S\|$ is also a *partial* equivalence relation on $|S|$, which shows that modest sets are in fact equivalent to PER models.

3 Theories and Signatures

In this section we describe first-order theories and signatures. Our system translates the former into the later.

3.1 Theories

A *theory* is a description of a mathematical structure, such as a group, a vector space, a directed graph, etc. A theory consists of

- a list of *basic sets*,
- a list of *basic constants* belonging to specified sets,
- a list of *basic relations* on specified sets,

[4] We do *not* require that a total value must be a terminating expression.

Theory Elements

`set` *s* [`=` *set*]
`const` *c* [`:` *set*] [`=` *term*]
[`stable`] `relation` *r* [`:` *set*] [`=` *prop*]
`equivalence` `:` *set*
`model` *M* `:` *theory*
`axiom` *a* [*M* `:` *theory*]* [*x* `:` *set*]* `=` *prop*

Propositions

`true`
`false`
`not` *prop*
prop `&&` *prop*
prop `||` *prop*
prop `=>` *prop*
prop `<=>` *prop*
r[*term*]*
term `=` *term*
`all` [*x* `:` *set*] `.` *prop*
`some` [*x* `:` *set*] `.` *prop*
`unique` [*x* `:` *set*] `.` *prop*

Sets

`0`
`1`
`bool`
s
Model `.` *name*
set `*` ⋯ `*` *set*
set `->` *set*
'label [`:` *set*] `+` ⋯ `+` *'label* [`:` *set*]
{ *x*[`:` *set*] `|` *proposition* }
set `%` *relation*

Terms

x
`(` *term* `,` ⋯ `,` *term* `)`
term `.` *n*
'label [*term*]
`match` *term* `with` *pattern-matches*
`lam` *x* `:` *set* `.` *term*
term term
term `%` *relation*
`let` *x* `%` *relation* `in` *term* `=` *term*
term `:>` *set*
term `:<` *set*
`the` *x* [`:` *set*] `.` *prop*
`let` *x* [`:` *set*] `=` *term* `in` *term*

Fig. 2. Input Language Summary

- a list of axioms.

To take a simple example, consider the theory of a semigroup in which every element has a (possibly non-unique) square root; recall that a semigroup is a set with an associative binary operation and a neutral element.[5] In our system it could be written as follows:

```
theory SQGROUP = thy
  set s
  const e : s
  const ( * ) : s -> s -> s
  implicit x, y, z : s
  axiom unit x      =  x * e = x and e * x = x
  axiom assoc x y z =  x * (y * z) = (x * y) * z
  axiom sqrt x      =  some y . y * y = x
end
```

The theory is enclosed by `thy`...`end`. This theory defines one basic set `s`, and two basic constants: an element `e` of `s` and a (curried) binary infix operator `*` on the set `s`. The `implicit` operator is not part of the theory proper, but

[5] An example of a semigroup with square roots is the complex numbers with multiplication as the binary operation.

signals to the type checker that bound variables named `x` or `y` or `z` should be assumed to range over `s` unless otherwise specified. Finally, we have three axioms. Axiom arguments, e.g., `x`, `y`, and `z` in the associativity axiom, name the free variables occuring in the axiom. It is not too big a mistake to think of them as being universally quantified.

It is important to note that theories do not include proofs, but rather just the statements of the axioms (and theorems) specified to hold. Thus although axioms can be defined, one cannot actually refer to them within the theory.

There are several features of theories that our system supports other than those shown in this example above; the input language is summarized in Figure 2, where brackets imply optional elements.

Theories may declare or define relations. They may be `stable`, i.e., their computational interpretation is trivial (see Section 4 for further discussion of this point). Axioms can universally quantify over all models of a theory. This is useful for describing universality properties, such as initiality of an algebra or finality of a coalgebra.

The propositions are the familiar ones from first-order logic; `unique` is unique existence (∃!). In addition to the basic empty (`0`) and unit (`1`) sets, one can form cartesian products, function spaces, tagged disjoint unions, subsets, and quotients by stable equivalence relations. The corresponding introduction and elimination forms appear in the language of terms. For example, *term* `%` *relation* is the equivalence class under *relation* containing *term*, while `let` x `%` *relation* `=` $term_1$ `in` $term_2$ binds x to a representative of the equivalence class $term_1$ to be used in $term_2$. The expression *term* `:>` *set* injects *term* into a given subset (recording a proof obligation of the term actually being a member of the subset), while *term* `:<` *set* projects *term* from a subset out into its superset *set*. The value of the description operator `the` x . *prop* is the unique x satisfying *prop*; using it incurs the obligation of proving that there is exactly one such x.

3.2 Signatures

On the logical side, we have models described by theories. Thus on the programming side we should have implementations being described by specifications. Our tool thus translates theories into *signatures*, which are ML's module interfaces.

Signatures allow us to require the existence of certain types, as well as values of given type. This allows decidable typechecking, but we need more expressiveness in order to faithfully translate the content of a theory. We therefore generate signatures augmented by assertion comments, which specify constraints on the values and functions an implementation beyond their type. It is the responsibility of the programmer to check that the implementation satisfies these assertions, as RZ does not attempt to do any theorem proving.

Assertions are written in ordinary classical first-order logic. Since pro-

grammers typically are not trained in constructive logic, this may make it easier to verify the assertions.

The output for the theory `SQGROUP` above is then:

```
module type SQGROUP =
sig
  type s
  (** Assertion per_s = PER(=s=) *)
  val e : s
  (** Assertion e_total =  e : ||s|| *)
  val ( * ) : s -> s -> s
  (** Assertion ( * )_total =
    all (x:s, y:s).  x =s= y =>
      all (x':s, y':s).  x' =s= y' =>
        x * x' =s= y * y'
  *)
  (**  Assertion unit (x:||s||) =
    x * e =s= x and e * x =s= x
  *)
  (**  Assertion
    assoc (x:||s||, y:||s||, z:||s||) =
      x * (y * z) =s= (x * y) * z
  *)
  val sqrt : s -> s
  (**  Assertion sqrt (x:||s||) =
    sqrt x : ||s||  and  sqrt x * sqrt x =s= x
  *)
end
```

At the ML level we have required a type `s`, and three values `e`, `*`, and `sqrt`, of types `s`, `s->s->s`, and `s->s`, respectively. The third value was generated from the square root axiom, which has a non-trivial computational content, cf. Subsection 4.2.

Comments contain other requirements, not expressible in ML, that further contstrain the allowed implementations. The assertion `PER(=s=)` abbreviates the requirement that `=s=` be a partial equivalence relation on `s`; its domain `||s||` is the subset of terms of type `s` that realize semigroup elements, and the relation `=s=` identifies (possibly different) terms realizing the same abstract semigroup element. These data together determine a modest set. The assertion following the declarations of `e` asserts that `e` realizes a valid semigroup element, and the one following `*` asserts that `*` must not be affected by the choice of realizers. Both `e` and `*` must of course still satisfy the `unit` and `assoc` axioms. Finally, the new function `sqrt` derived from the logic must compute square roots. Since the theory requires existence but not uniqueness of square roots, there is no requirement that `sqrt` be invariant with respect to the partial equivalence relation on `s`; different realizers of the same semigroup

element are allowed to produce (realizers of) different square roots.

3.3 Parameterized Theories

A theory may be parameterized by one or more models of other theories. For example, a theory `Real` of the reals may be parameterized in terms of a model `N` of the naturals. A theory of free groups may be parameterized in terms of the generating set.

Parameterized theories serve two purposes. A model of a parameterized theory is a generic implementation that, given any implementation of the parameters, returns an implementation of the resulting theory. At the level of ML, this would be a function from modules to modules, a so-called *functor*, and so a parameterized theory can be translated into the signature of a functor.

Alternatively, once we have described a parameterized theory `Real`, we may wish to use it to describe a single specific implementation of real numbers based on a specific model `N1` (implementation) of the natural numbers; this can be described as an implementation satisfying the theory `Real(N1)`.

The dual nature of parameterized theories as being both a description of a parameterized model (a Π type) and something which can be applied to a model to produce a specialized theory (a λ) is very reminiscent the type inclusion of Automath [2]. ML does not permit applications of functor signatures, however, so we beta-reduce all theory applications before generating signatures; `Real(N1)` would produce a signature for a real-number implementation that refers directly to `N1` rather than to a generic parameter `N`.

4 Implementation

4.1 Pre-translation

After parsing, our implementation does type checking. The type checker does simple type reconstruction. Instead of doing full unification, we require that the types of all bound variables must either be given at the binding site, given through an `implicit` declaration, or be obvious from their definition.

Unlike ML datatypes, we do not require disjoint union type to be declared before they are used, or to have different unions involve different tags. Therefore, a very small amount of implicit subtyping is done between sums. Otherwise, if we had a specifiation of queues of integers that included

```
set iqueue
const emptyQueue
const enqueue : int*iqueue -> iqueue
const dequeue : iqueue -> `None + `Some:int*iqueue
```

then the axiom

```
dequeue emptyQueue = `None
```

would fail to typecheck (the most-precise set for the left-hand-side is a two-element disjoint unition, while the most-precise set for the right is a one-element disjoint union.)

The type checker will also try to convert between *set* and a subset type { *x* : *set* | *proposition* } as necessary in order to type check. Thus,

```
set real
set nz_real = {x:real | not (x=zero)}
const one   : real
const inv   : nz_real -> nz_real
const ( * ) : real * real -> real

axiom field (x : real) =
  not (x=zero)  =>  x * (inv x) = one
```

is allowed, instead of requiring

```
axiom field (x : real) =
  not (x=zero)  =>  x * ((inv (x:>nz_real)) :< real) = one
```

In this case, since `:>` and `:<` has computational content (going into a subset involves pairing the item with the realizer of the proposition; going out of a subset is then a first projection), the typechecker rewrites the former version of `field` into the latter before passing it on to the translation phase. If injections into subsets cannot be justified in the theory (e.g., if the `field` axiom lacked the premise `not (x=zero)`) then the theory will still translate, but the generated assertions will not be satisfied by any implementation.

4.2 Realizability Translation

We first discuss the realizability translation of sets and terms, and then focus on the translation of logic, which is the interesting part.

A set S is translated into a modest set $(|S|, \|S\|, \approx_S)$ according to its structure: a basic set is translated to a modest set whose underlying type is abstract, a product is translated to a product of modest sets, a function space is translated to the exponential of modest sets, etc. Thus we simply use the rich categorical structure of $\mathsf{Mod}(\mathcal{P})$ to interpret all the set constructors.

Similarly, terms are translated to suitable ML terms according to their structure. Note however, that there are terms whose validity cannot be checked by RZ because that would require it to prove arbitrary theorems. Such an example is the definite description operator `the` $x\,.\,\phi(x)$, whose validity can be confirmed only by proving that there exists a unique x satisfying $\phi(x)$. In such cases RZ emits proof obligations for the programmer to verify. Note however, that the translated terms always have valid ML types, even if the accompanying proof obligations are not satisfied.

The driving force behind the realizability translation of logic is a theorem, see e.g. [12, Thm. 4.4.10], which says that under the realizability interpretation

every formula ϕ is equivalent to one that says, informally speaking, "there exists r, such that r realizes ϕ". Furthermore, the formula "r realizes ϕ" is computationally trivial. We explain what precisely this means next.

In classical logic a doubly negated formula $\neg\neg\phi$ is equivalent to ϕ. Constructively, this is not true in general. To see this, recall that in constructive logic $\neg\phi$ is defined as $\phi \implies \bot$ and observe that the underlying type of realizers of $\neg\neg\phi$ is $(|\phi| \to \texttt{unit}) \to \texttt{unit}$. Terms of this type cannot be converted to terms of type $|\phi|$ in a general way (although conversion in the reverse direction can be done quite easily, which shows that ϕ implies $\neg\neg\phi$). Furthermore, terms of type $(|\phi| \to \texttt{unit}) \to \texttt{unit}$ do not compute much of anything, so we might as well replace them by a special *trivial realizer* devoid of any computational meaning. We can think of the trivial realizer as a term which witnesses validity of a formula but does not compute anything.

In some cases it may happen even in constructive logic that $\neg\neg\phi$ is equivalent to ϕ. When this is so we call ϕ a *$\neg\neg$-stable formula*, or just *stable formula* for short. Stable formulas have trivial realizers, as they are equivalent to doubly negated formulas. Among the stable formulas the *almost negative* formulas are important because they can be easily recognized syntactically: they are built from any combination of $\wedge$, $\implies$, $\forall$, $=$, and those basic predicates that are known to be stable, but $\exists$ and $\vee$ are only allowed to appear on the left side of an $\implies$.[6]

The following theorem is a precise formulation of the claims we made in a paragraph above.

Theorem 4.1 *For every formula ϕ there exists a set R_ϕ and an almost negative formula ϕ' such that in the realizability interpretation ϕ is equivalent to $\exists r \in R_\phi.\, \phi'(r)$.*

We omit the proof, as it is fairly standard and involves a straightforward induction on the structure of ϕ. The set R_ϕ in the theorem is simply the set of terms of the underlying type $|\phi|$ of realizers, while the intuitive meaning of $\phi'(r)$ is "r realizes ϕ".

RZ translates an axiom ϕ, or any other proposition it encounters, by computing its underlying type $|\phi|$ and the almost negative formula ϕ' from the above theorem. In the output signature it then emits

$$\texttt{val}\ r : |\phi|$$
$$(*\ \text{Assertion}\ \phi'(r)\ *)$$

This way the axiom ϕ has been separated into its computational content r and a statement ϕ' which describes when r is a valid realizer of ϕ. Because ϕ' is almost negative it has no computational content, which means that its classical and constructive readings agree. Therefore a constructive mathematician and a classical programmer will agree on the meaning of $\phi'(r)$.

[6] A *negative* formula is one that does not contain $\exists$ and $\vee$ at all.

RZ recognizes almost negative formulas and optimizes away their realizers, as described below. In addition, the user may declare a basic predicate or relation to be stable, which will be taken into account by RZ during translation and optimization.

It seems worth noting that the computational irrelevance of stable propositions is akin to *proof irrelevance* studied by Pfenning [9]. This is not surprising in view of the well known fact that double negation enjoys many formal properties of a modal possibility operator.

4.3 Optimization

Propositions without constructive content have trivial realizers, and so a final "unit elimination" pass both removes these and does peephole simplification of the resulting signature. Without an optimizer, the axioms of the theory `SQGROUP` would produce

```
val unit : s -> top * top
(** Assertion unit (x:||s||) =
     x * e =s= x and e * x =s= x
*)

val sqrt : s -> s * top
(** Assertion sqrt (x:||s||) =
     pi0(sqrt x) : ||s|| and
     pi0(sqrt x) * pi0(sqrt x) =s= x
*)
```

where `top` is the type of trivial realizers; we use `top` instead of `unit` to emphasize that these trivial realizers are terminating and hence safe to eliminate; this would not necessarily be true for terms of type `unit`. The optimizer can easily tell from the types that the realizers for the `unit` and `assoc` axioms are trivial and can be discarded, and that although `sqrt` cannot be discarded entirely, part of its return value is unnecessary. Assertions that reference discarded or optimized constants are automatically rewritten to preserve well-typedness, and we obtain the translation of `SQGROUP` shown previously, which contains no occurrences of `top`.

5 Examples

Decidable set.

We now consider the theory of a decidable set. Recall that in constructive mathematics a set S is said to be *decidable* if $x = y$ or $x \neq y$ for all $x, y \in S$. The input to RZ is

```
theory DecidableSet = thy
  set s
```

```
  axiom decidable (x:s) (y:s)  =  (x = y) or not (x = y)
end
```

and the output is

```
module type DecidableSet =
sig
  type s
  (** Assertion per_s = PER(=s=) *)

  val decidable : s -> s -> [‘or0 | ‘or1]
  (** Assertion
    decidable (x:||s||, y:||s||) =
      decidable x y = ‘or0 and x =s= y cor
      decidable x y = ‘or1 and not (x =s= y)
  *)
end
```

The output signature asks for `decidable` to be a function accepting two realizers x and y and returning one of two tokens `‘or0` and `‘or1`, depending on whether x and y realize the same element. (Disjunction is written `cor` to emphasize that it is classical or.) This is nothing but a computable decision procedure for equality on `s` with respect to `=s=`, as should be expected.

We remark that nothing requires the partial equivalence relation `=s=` to be computable, so not every modest set is decidable. In fact, there are many natural and important examples of non-computable partial equivalence relations, such as *extensional equality* of functions from natural numbers to natural numbers. (If we could computably decide whether two functions always give equal results on equal arguments, we could construct a Halting Oracle.)

Natural Numbers.

Next we consider the theory of natural numbers. This example shows how axioms can be parameterized by theories. Recall that the natural numbers are the initial algebra with one constant and one unary operation (such algebras are sometimes called “iteration algebras”):

```
theory Iteration = thy
 set s
 const zero : s
 const succ : s -> s
end

theory Nat = thy
 model N : Iteration

 axiom initial [I : Iteration] =
   unique (f : N.s -> I.s).
```

```
module type Iteration =
sig
 type s
 (**  Assertion per_s =  PER(=s=) *)
 val zero : s
 (**  Assertion zero_total =  zero : ||s|| *)
 val succ : s -> s
 (**  Assertion succ_total =
   all (x:s, y:s).  x =s= y => succ x =s= succ y *)
end

module type Nat =
sig
 module N : Iteration
 module Initial : functor (I : Iteration) ->
  sig
   val initial : N.s -> I.s
    (**  Assertion initial =
      (all (x:N.s, y:N.s).  x =N.s= y => initial x =I.s= initial y) and
      initial N.zero =I.s= I.zero and
      (all (n:||N.s||). initial (N.succ n) =I.s= I.succ (initial n)) and
      (all (u:N.s -> I.s).
        (all (x:N.s, y:N.s).  x =N.s= y => u x =I.s= u y) =>
        u N.zero =I.s= I.zero and
        (all (n:||N.s||). u (N.succ n) =I.s= I.succ (u n)) =>
        all (x:N.s, y:N.s).  x =N.s= y => initial x =I.s= u y)
     *)
 end
end
```

Fig. 3. Output for theories `Iteration` and `Nat`

```
    (f N.zero = I.zero  and  all (n : N.s) . f (N.succ n) = I.succ (f n))
end
```

The theory `Iteration` is an auxiliary theory. The theory `Nat` postulates the existence of a model `N` of theory `Iteration` which satisfies the initiality axiom stating that there exists exactly one algebra morphism from `N` to any other iteration theory `I`. The output generated by RZ is shown in Figure 3. The initiality axiom has been translated to a functor which expects an implementation `I` of an iteration theory and outputs a realizer for the axiom. A closer look at the assertion reveals that it essentially says that the realizer defines a function from natural numbers to `I.s` by simple recursion.

Axiom of Choice.

As a third example, we look at the realizability interpretation of the Axiom of Choice. We work with the formulation of the axiom which states that every

total relation has a choice function:

$$(\forall x \in A.\, \exists y \in B.\, R(x,y)) \implies \exists g \in B^A.\, \forall x \in A.\, R(x, g(x)) \ .$$

We could write this as a theory parameterized by sets A, B and the relation R, but to keep things simple, we use the following version:

```
theory Choice = thy
 set a
 set b
 relation r : a * b
 axiom choice = (all (x:a). some (y:b) . r(x,y)) =>
                   some (g:a->b) . all (x:a) . r(x,g(x))
end
```

The output is shown in Figure 4. The interesting bit is the assertion for `choice`, which says that `choice` takes as an argument a realizer f for the $\forall\exists$ statement and outputs a pair of functions, of which the first is the choice function g and the second one provides realizers witnessing that the choice function does its job. However, there is a problem: the realizer f is not required to respect `=a=` while the choice function g is. In general there is no way for `choice` to transform f into a `=a=`-respecting function. It follows that in general the Axiom of Choice is *not* valid in the realizability interpretation. This is another important difference between realizability and propositions-as-types.

6 Conclusions and Future Work

By translating only at the level of specifications, RZ provides a useful middle ground between ad-hoc implementations and machine-generated implementations. It allows much more flexible implementation strategies, but relies on programmers to verify properties of their code.

Further, RZ can serve as a means of explaining constructive mathematics to programmers. Programmers who are not knowledgeable about constructive mathematics can still understand the output of the translation, which involves familiar concepts such as abstract types and (classical) first-order logic. Looking at such examples can provide the necessary intuition behind the original logic, and better explain why one might want to work with constructive rather than classical logic to begin with.

Axioms parameterized by models (e.g., initiality) currently translate into signatures of ML functors. We have experimented with an alternative translation of such axioms into polymorphic types. In this case the `initial` axiom of the natural numbers yields the specification

```
val initial: 'a -> ('a -> 'a) -> N.s -> 'a
```

which is exactly the familiar iterator for natural numbers (i.e., given an initial value, a function, and a natural number, apply the function that many times

```
module type Choice =
sig
  type a
  (** Assertion per_a =  PER(=a=) *)
  type b
  (** Assertion per_b =  PER(=b=) *)
  type r
  (** Assertion predicate_r =
    PREDICATE(r, a * b,
      lam t u.(pi0 t =a= pi0 u and pi1 t =b= pi1 u))
  *)

  val choice : (a -> b * r) -> (a -> b) * (a -> r)
  (**  Assertion choice =
    all (f:a -> b * r).
      (all (x:||a||).  pi0 (f x) : ||b|| and
         pi1 (f x) |= r (x,pi0 (f x))) =>
      (all (x:a, y:a).  x =a= y => pi0 (choice f) x =b= pi0 (choice f) y) and
      (all (x:||a||).  pi1 (choice f) x |= r (x,pi0 (choice f) x))
  *)
end
```

Fig. 4. Output for theory `Choice`

to the initial value). Such types can be much more natural and much simpler for programmers to understand. The theory behind the translation is well understood, being the phase-splitting translation of Harper, Mitchell, and Moggi [3]. Because of limitations of ML not every parameterized axiom can be turned into polymorphism; ML allows only prenex quantifiers, and the quantifiers can range over types but not type operators. However we would like to do so where it is possible (the common case). As an alternative, we could attempt to retarget the output to a language like Haskell [10] which supports the necessary polymorphic types, though Haskell's support of modules is much weaker.

Another possible extension would be to allow dependent families in the input language. Fortunately, this does not require finding a target language that supports dependent types; we can use the underlying (non-dependent) types, and then express the dependencies as additional properties that must be verified for the implementation.

References

[1] A. Bauer. *The Realizability Approach to Computable Analysis and Topology.* PhD thesis, Carnegie Mellon University, 2000. Available as CMU technical report `CMU-CS-00-164` and at `http://andrej.com/thesis`.

[2] Nicolas G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.

[3] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order Modules and the Phase Distinction. In *Proc. 17th ACM Symposium on Principles of Programming Languages (POPL '90)*, pages 341–354, 1990.

[4] J.M.E. Hyland. The effective topos. In A.S. Troelstra and D. Van Dalen, editors, *The L.E.J. Brouwer Centenary Symposium*, pages 165–216. North Holland Publishing Company, 1982.

[5] J.M.E. Hyland, P.T. Johnstone, and A.M. Pitts. Tripos theory. *Math. Proc. Camb. Phil. Soc.*, 88:205–232, 1980.

[6] S.C. Kleene. On the interpretation of intuitionistic number theory. *Journal of Symbolic Logic*, 10:109–124, 1945.

[7] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system, documentation and user's manual - release 3.08. Technical report, INRIA, July 2004.

[8] John Longley. Matching typed and untyped realizability. *Electr. Notes Theor. Comput. Sci.*, 35, 2000.

[9] F. Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS'01)*, page 221. IEEE Computer Society, June 2001.

[10] Simon Peyton Jones, ed. The Haskell 98 language. *Journal of Functional Programming*, 13(1), January 2003.

[11] A.S. Troelstra. Realizability. In S.R. Buss, editor, *Handbook of Proof Theory*, pages 407–473. North-Holland, 1998.

[12] A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics, An Introduction, Vol. 1.* Number 121 in Studies in Logic and the Foundations of Mathematics. North-Holland, 1988.

[13] J. van Oosten. *Exercises in Realizability.* PhD thesis, Universiteit van Amsterdam, 1991.