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

Andrej Bauer[1] and Christopher A. Stone[2]

[1] Faculty of Mathematics and Physics, University of Ljubljana, Slovenia
Andrej.Bauer@fmf.uni-lj.si
[2] Computer Science Department, Harvey Mudd College, USA
stone@cs.hmc.edu

Abstract. Realizability theory can produce interfaces for the data structure corresponding to a mathematical theory. Our tool, called RZ, serves as a bridge between constructive mathematics and programming by translating specifications in constructive logic into annotated interface code in Objective Caml. The system supports a rich input language allowing descriptions of complex mathematical structures. RZ does not extract code from proofs, but allows any implementation method, from handwritten code to code extracted from proofs by other tools.

Version of January 29, 2007.

1 Introduction

Given a description of a mathematical structure (constants, functions, relations, and axioms), what should a computer implementation look like?

For simple cases, like groups, the answer is obvious. But for more interesting structures, especially those arising in mathematical analysis, the answer is less clear. How do we implement the real numbers (a Cauchy-complete Archimedean ordered field)? Or choose the operations for a compact metric space or a space of smooth functions? Significant research goes into finding satisfactory representations [1–4], and implementations of exact real arithmetic [5, 6] show that the theory can be put into practice quite successfully.

Realizability theory can be used to produce a description of the data structure (a code interface) directly corresponding to a mathematical specification. But few programmers — even those with strong backgrounds in mathematics and classical logic — are familiar with constructive logic or realizability.

We have therefore implemented a system, called RZ, to serve as a bridge between the logical world and the programming world.[3] RZ translates specifications in constructive logic into standard interface code in a programming language (currently Objective Caml [7], but other languages could be used).

[3] RZ is publicly available for download at `http://math.andrej.com/rz/`, together with an extended version of this paper.

The constructive part of the original specification turns into interface code, listing types and values to be implemented. The rest becomes assertions about these types and values. The assertions have no computational content, so their constructive and classical meanings agree, and they can be understood by programmers and mathematicians accustomed to classical logic.

RZ was designed as a lightweight system supporting a rich input language. Although transforming complete proofs into complete code is possible [8], we have not implemented this. Other good systems, including Coq [9] and Minlog [10], can extract programs from proofs. But they work best managing the entire task, from specification to code generation. In contrast, interfaces generated by RZ can be implemented in any fashion as long as the assertions are satisfied. Code can be written by hand, using imperative, concurrent, and other language features rather than a "purely functional" subset. Or, the output can serve as a basis for theorem-proving and code extraction using another system.

2 Typed realizability

RZ is based on *typed realizability* by John Longley [11]. This variant of realizability corresponds most directly to programmers' intuition about implementations.

We approach typed realizability and its relationship to real-world programming by way of example. Suppose we are asked to design a data structure for the set $\mathcal{G}$ of all finite simple directed graphs with vertices labeled by distinct integers. A common representation is a pair of lists (ℓ_V, ℓ_A), where ℓ_V is the list of vertex labels and ℓ_A is the *adjacency list* representing the arrows by pairing the labels of each source and target. Thus we define the datatype of graphs as[4]

$$\mathtt{type}\ \mathtt{graph} = \mathtt{int}\ \mathtt{list} \quad * \quad (\mathtt{int} * \mathtt{int})\ \mathtt{list}$$

However, this is not a complete description of the representation, as there would be representation invariants and conditions not expressed by the type, e.g., the order in which vertices and arrows are listed is not important, each vertex and arrow must be listed exactly once, and the source and target of each arrow must appear in the list of vertices.

Thus, to implement the mathematical set $\mathcal{G}$, we must not only decide on the underlying datatype `graph`, but also determine what values of that type represent which elements of $\mathcal{G}$. As we shall see next, this can be expressed either using a *realizability relation* or a *partial equivalence relation (per).*

2.1 Modest sets and pers

We now define typed realizability as it applies to OCaml. Other general-purpose programming languages could be used instead.

---

[4] We use OCaml notation in which t `list` classifies finite lists of elements of type t, and $t_1 * t_2$ classifies pairs containing a value of type t_1 and and value of type t_2.

Let Type be the collection of all (non-parametric) OCaml types. To each type $t \in \mathsf{Type}$ we assign the set $[\![t]\!]$ of values of type t that behave *functionally* in the sense of Longley [12]. Such values are represented by terminating expressions that do not throw exceptions or return different results on different invocations. They may *use* exceptions, store, and other computational effects, provided they appear functional from the outside; a useful example using computational effects is presented in Section 7.4. A functional value of function type may diverge as soon as it is applied. The collection Type with the assignment of functional values $[\![t]\!]$ to each $t \in \mathsf{Type}$ forms a *typed partial combinatory algebra (TPCA)*.

Going back to our example, we see that an implementation of directed graphs $\mathcal{G}$ specifies a datatype $|\mathcal{G}| = \texttt{graph}$ together with a *realizability relation* $\Vdash_{\mathcal{G}}$ between $\mathcal{G}$ and $[\![\texttt{graph}]\!]$. The meaning of $(\ell_V, \ell_A) \Vdash_{\mathcal{G}} G$ is "OCaml value (ℓ_V, ℓ_A) represents/realizes/implements graph G". Generalizing from this, we define a *modest set* to be a triple $A = (\langle A \rangle, |A|, \Vdash_A)$ where $\langle A \rangle$ is the *underlying set*, $|A| \in \mathsf{Type}$ is the *underlying type*, and $\Vdash_A$ is a *realizability relation* between $[\![|A|]\!]$ and $\langle A \rangle$, satisfying (1) *totality:* for every $x \in \langle A \rangle$ there is $v \in [\![|A|]\!]$ such that $v \Vdash_A x$, and (2) *modesty:* if $u \Vdash_A x$ and $u \Vdash_A y$ then $x = y$. The *support* of A is the set $\|A\| = \{v \in [\![|A|]\!] \mid \exists x \in \langle A \rangle \,.\, v \Vdash_A x\}$ of those values that realize something. We define the relation $\approx_A$ on $[\![|A|]\!]$ by

$$u \approx_A v \iff \exists x \in \langle A \rangle \,.\, (u \Vdash_A x \wedge v \Vdash_A x) \ .$$

From totality and modesty of $\Vdash_A$ it follows that $\approx_A$ is a per, i.e., symmetric and transitive. Observe that $\|A\| = \{v \in [\![|A|]\!] \mid v \approx_A v\}$, whence $\approx_A$ restricted to $\|A\|$ is an equivalence relation. In fact, we may recover a modest set up to isomorphism from $|A|$ and $\approx_A$ by taking $\langle A \rangle$ to be the set of equivalence classes of $\approx_A$, and $v \Vdash_A x$ to mean $v \in x$.

The two views of implementations, as modest sets $(\langle A \rangle, |A|, \Vdash_A)$, and as pers $(|A|, \approx_A)$, are equivalent.[5] We concentrate on the view of modest sets as pers. They are more convenient to use in RZ because they refer only to types and values, as opposed to arbitrary sets. Nevertheless, it is useful to understand how modest sets and pers arise from natural programming practice.

Pers form a category whose objects are pairs $A = (|A|, \approx_A)$ where $|A| \in \mathsf{Type}$ and $\approx_A$ is a per on $[\![|A|]\!]$. A morphism $A \to B$ is represented by a function $v \in [\![|A| \to |B|]\!]$ such that, for all $u, u' \in \|A\|$, $u \approx_A u' \implies v\ u \approx_B v\ u'$. Two such functions v and v' represent the same morphism if, for all $u, u' \in \|A\|$, $u \approx_A u'$ implies $v\ u \approx_B v'\ u'$.

The category of pers has a very rich structure, namely that of a regular locally cartesian closed category [13]. This suffices for the interpretation of first-order logic and (extensional) dependent types [14].

Not all pers are *decidable*, i.e., there may be no algorithm for deciding when two values are equivalent. Examples include implementations of semigroups with an undecidable word problem [15] and implementations of computable real numbers (which might be realized by infinite Cauchy sequences).

[5] And there is a third view, as a partial surjection $\delta_A : \subseteq [\![|A|]\!] \twoheadrightarrow \langle A \rangle$, with $\delta_A(v) = x$ when $v \Vdash_A x$. This is how realizability is presented in Type Two Effectivity [1].

Underlying types of realizers:

$$
\begin{array}{llll}
|\top| &= \texttt{unit} & |\bot| &= \texttt{unit} \\
|x = y| &= \texttt{unit} & |\phi \wedge \psi| &= |\phi| \times |\psi| \\
|\phi \Rightarrow \psi| &= |\phi| \to |\psi| & |\phi \vee \psi| &= `\texttt{or}_0\ \texttt{of}\ |\phi_0| + `\texttt{or}_1\ \texttt{of}\ |\phi_1| \\
|\forall x{:}A.\ \phi| &= |A| \to |\phi| & |\exists x{:}A.\ \phi| &= |A| \times |\phi|
\end{array}
$$

Realizers:

$$
\begin{array}{lll}
() \Vdash \top & & \\
() \Vdash x = y & \text{iff} & x = y \\
(t_1, t_2) \Vdash \phi \wedge \psi & \text{iff} & t_1 \Vdash \phi \text{ and } t_2 \Vdash \psi \\
t \Vdash \phi \Rightarrow \psi & \text{iff} & \text{for all } u \in |\phi|, \text{ if } u \Vdash \phi \text{ then } t\,u \Vdash \psi \\
`\texttt{or}_0\, t \Vdash \phi \vee \psi & \text{iff} & t \Vdash \phi \\
`\texttt{or}_1\, t \Vdash \phi \vee \psi & \text{iff} & t \Vdash \psi \\
t \Vdash \forall x{:}A.\ \phi(x) & \text{iff} & \text{for all } u \in |A|, \text{ if } u \Vdash_A x \text{ then } t\,u \Vdash \phi(x) \\
(t_1, t_2) \Vdash \exists x{:}A.\ \phi(x) & \text{iff} & t_1 \Vdash_A x \text{ and } t_2 \Vdash \phi(x)
\end{array}
$$

Fig. 1. Realizability interpretation of logic (outline)

2.2 Interpretation of logic

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 proofs of a proposition correspond to terms of the corresponding type. More precisely, to each formula ϕ we assign an underlying type $|\phi|$ of realizers, but unlike the propositions-as-types translation, not all terms of type $|\phi|$ are necessarily valid realizers for ϕ, and some terms that are realizers may not correspond to any proofs, for example, if they denote partial functions or use computational effects.

It is customary to write $t \Vdash \phi$ when $t \in [\![|\phi|]\!]$ is a realizer for ϕ. 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* if it has at least one realizer.

In classical mathematics, a predicate on a set X may be viewed as a subset of X or a (possibly non-computable) function $X \to \texttt{bool}$, where $\texttt{bool} = \{\bot, \top\}$ is the set of truth values. Accordingly, since in realizability propositions are witnessed by realizers, a predicate ϕ on a per $A = (|A|, \approx_A)$ is a (possibly non-computable) function $\phi : [\![|A|]\!] \times [\![|\phi|]\!] \to \texttt{bool}$ that is *strict* (if $\phi(u, v)$ then $u \in \|A\|$) and *extensional* (if $\phi(u_1, v)$ and $u_1 \approx_A u_2$ then $\phi(u_2, v)$).

Suppose we have implemented the real numbers $\mathbb{R}$ as a per $R = (\texttt{real}, \approx_R)$, and consider $\forall a{:}R.\ \forall b{:}R.\ \exists x{:}R.\ x^3 + ax + b = 0$. By computing according to Figure 1, we see that a realizer for this proposition is a value r of type $\texttt{real} \to \texttt{real} \to \texttt{real} \times \texttt{unit}$ such that, if t realizes $a \in \mathbb{R}$ and u realizes $b \in \mathbb{R}$, then $r\,t\,u = (v, w)$ with v realizing a real number x such that $x^3 + ax + b = 0$, and w is trivial. (This can be "thinned" to a realizer of type $\texttt{real} \to \texttt{real} \to \texttt{real}$ that does not bother to compute w.) In essence, the realizer r computes a root of the cubic equation. Note that r is *not* extensional, i.e., different realizers t and u for the same a and b may result in different roots. To put this in another way, r realizes a *multi-valued* function[6] rather than a per morphism. It is well known in computable mathematics that certain operations, such as equation solving, are only computable if we allow them to be multi-valued. They arise naturally in RZ as translations of $\forall\exists$ statements.

Some propositions, such as equality and negation, have "irrelevant" realizers free of computational content. Sometimes only a part of a realizer is computationally irrelevant. Propositions that are free of computational content are characterized as the *$\neg\neg$-stable propositions*. A proposition ϕ is said to be $\neg\neg$-stable, or just *stable* for short, when $\neg\neg\phi \Rightarrow \phi$ is valid. On input, one can specify whether abstract predicates have computational content. On output, extracted realizers go through a *thinning* phase, which removes irrelevant realizers.

Many structures are naturally viewed as families of sets, or sets depending on parameters, or *dependent types* as they are called in type theory. For example, the n-dimensional Euclidean space $\mathbb{R}^n$ depends on the dimension $n \in \mathbb{N}$, the Banach space $\mathcal{C}([a, b])$ of uniformly continuous real functions on the closed interval $[a, b]$ depends on $a, b \in \mathbb{R}$ such that $a < b$, etc. In general, a family of sets $\{A_i\}_{i \in I}$ is an assignment of a set A_i to each $i \in I$ from an *index set* I.

In the category of pers the appropriate notion is that of a *uniform* family. A uniform family of pers $\{A_i\}_{i \in I}$ indexed by a per I is given by an underlying type $|A|$ and a family of pers $(\approx_{A_i})_{i \in [\![|I|]\!]}$ that is strict (if $u \approx_{A_i} v$ then $i \in \|I\|$) and extensional (if $u \approx_{A_i} v$ and $i \approx_I j$ then $u \approx_{A_j} v$).

We can also form the *sum* $\Sigma_{i \in I} A_i$ or *product* $\Pi_{i \in I} A_i$ of a uniform family, allowing an interpretation of (extensional) dependent type theory.

3 Specifications as signatures with assertions

In programming we distinguish between *implementation* and *specification* of a structure. In OCaml these two notions are expressed with modules and module types, respectively. A module defines types and values, while a module type simply lists the types, type definitions, and values provided by a module. For a complete specification, a module type must also be annotated with *assertions* which specify the required properties of declared types and values.

The output of RZ consists of *module specifications*, each of which consists of a module type plus assertions about its components. More specifically, a typical specification may contain value declarations, type declarations and definitions, module declarations, specification definitions, proposition declarations, and assertions. The language of specifications is summarized in Figure 3.

The least familiar construct is the *obligation* `assure` $x{:}\tau,\ p$ `in` e which means "in term e, let x be any element of $[\![\tau]\!]$ that satisfies p". An obligation is equivalent to a combination of Hilbert's indefinite description operator and a local

[6] The multi-valued nature of the realizer comes from the fact that it computes *any one* of many values, not that it computes *all* of the many values.

Types

$\tau ::= T \mid M.T$	Type names
$\mid \texttt{unit} \mid \tau_1 \times \tau_2$	Unit and cartesian product
$\mid \tau_1 \to \tau_2$	Function type
$\mid \text{`}l_1\ \texttt{of}\ \tau_1 + \cdots + \text{`}l_n\ \texttt{of}\ \tau_n$	Disjoint sum
$\mid \alpha$	Polymorphic types

Terms

$e ::= x \mid M.x$	Term names
$\mid \texttt{fun}\, x{:}\tau_1 \to e \mid e_1\, e_2$	Functions and application
$\mid () \mid (e_1, \ldots, e_n) \mid \texttt{p}_n\, e$	Tuples and projection
$\mid \text{`}l\, e \mid (\texttt{match}\ e\ \texttt{with}\ \text{`}l_1\, x_1 \to e_1 \mid \cdots \mid \text{`}l_n\, x_n \to e_n)$	Injection and projection from a sum
$\mid \texttt{assure}\, x{:}\tau,\ p\ \texttt{in}\ e \mid \texttt{assure}\ p\ \texttt{in}\ e$	Obligations
$\mid \texttt{let}\ x{=}e_1\ \texttt{in}\ e_2$	Local definitions

Propositions (negative fragment)

$p ::= P \mid M.P$	Atomic proposition
$\mid \top \mid \bot \mid \neg p_1 \mid p_1 \wedge p_2 \mid p_1 \Rightarrow p_2 \mid p_1 \Leftrightarrow p_2$	Predicate logic
$\mid \texttt{match}\ e\ \texttt{with}\ \text{`}l_1\, x_1 \Rightarrow p_1 \mid \cdots \mid \text{`}l_n\, p_n \Rightarrow e_n$	Propositional case
$\mid \texttt{fun}\, x{:}\tau \to p \mid p\, e$	Propositional functions and application
$\mid e_1 \approx_s e_2 \mid e{:}\Vert s \Vert$	Pers and support
$\mid e_1{=}e_2$	(Observational) term equality
$\mid \forall x{:}\tau.\, p \mid \forall x{:}\Vert s \Vert.\, p$	Term quantifiers

Basic modest sets

$s ::= S \mid s\, e$

Modules

$m ::= M \mid m.M$	Model names
$\mid m_1\, m_2$	Application of parameterized model

Proposition Kinds

$\Pi ::= \texttt{bool}$	Classifier for propositions
$\mid \tau \to \Pi$	Classifier for a predicate/relation

Specification elements

$\theta ::= \texttt{val}\ x : \tau$	Value declaration
$\mid \texttt{type}\ T$	Type declaration
$\mid \texttt{type}\ T = \tau$	Type definition
$\mid \texttt{module}\ m : \Sigma$	Module declaration
$\mid \texttt{module type}\ S = \Sigma$	Specification definition
$\mid \texttt{predicate}\ P : \Pi$	Proposition declaration
$\mid \texttt{assertion}\ A : p$	Assertion

Specifications (module types with assertions)

$\Sigma ::= S \mid m.S$	Specification names
$\mid \texttt{sig}\ \theta_1 \ldots \theta_n\ \texttt{end}$	Specification elements
$\mid \texttt{functor}\,(m{:}\Sigma_1) \to \Sigma_2$	Parameterized specification

Fig. 2. The syntax of specifications (Simplified)

definition, $\texttt{let}\, x{=}(\varepsilon x{:}\tau.\, p)\, \texttt{in}\, e$, where $\varepsilon x{:}\tau.\, p$ means "any $x \in [\![\tau]\!]$ such that p". The alternative form $\texttt{assure}\, p\, \texttt{in}\, e$ stands for $\texttt{assure}\ \_{:}\texttt{unit},\ p\ \texttt{in}\, e$.

Obligations arise from the fact that well-formedness of the input language is undecidable; see Section 4. In such cases the system computes a realizability translation, but also produces obligations. The programmer must replace each obligation with a value satisfying the obligation. If such values do not exist, the specification is unimplementable.

4 The Input Language

The input to RZ consists of one or more theories. A RZ *theory* is a generalized logical signature with associated axioms, similar to a Coq module signature. Theories describe *models*, or implementations. A summary of the input language appears in Figure 4.

The term language includes introduction and elimination constructs for the set level. For product sets we have tuples and projections ($\pi_1\, e$, $\pi_2\, e$, $\dots$), and for function spaces we have lambda abstractions and application. One can inject a term into a tagged union, or do case analyses on the members of a union. We can produce an equivalence class or pick a representative from a equivalence class (as long as what we do with it does not depend on the choice of representative). We can produce a set of realizers or choose a representative from a given set of realizers (as long as what we do with it does not depend on the choice of representative). We can inject a term into a subset (if it satisfies the appropriate predicate), or project an element out of a subset. Finally, the term language also allows local definitions of term variables, and definite descriptions (as long as there is a unique element satisfying the predicate in question).

From the previous paragraph, it is clear that checking the well-formedness of terms is not decidable. RZ checks what it can, but does not attempt serious theorem proving. Uncheckable constraints remain as obligations in the final output, and should be verified by other means before the output can be used.

5 Translation

Having shown the input and output languages for RZ, we briefly sketch the translation from one to the other. A theory is translated to a specification, where the theory elements are translated as follows.

5.1 Translation of sets and terms

A set declaration `Parameter` s : `Set` is translated to

```
type s
predicate (≈s) : s → s → bool
assertion symmetric_s :  ∀ x:s, y:s, x ≈s y -> y ≈s x
assertion transitive_s : ∀ x:s, y:s, z:s, x ≈s y ∧ y ≈s z → x ≈s z
predicate ‖s‖ : s → bool
assertion support_def_s :  ∀ x:s,  x : ‖s‖ ⇔ x ≈s x
```

Propositions

$\varphi, \rho ::= p \mid M.p$	Predicate names
$\mid \top \mid \bot \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \Rightarrow \varphi_2$	Predicate logic
$\mid \texttt{match}\, e\, \texttt{with}\, l_1\, x_1 \Rightarrow \varphi_1 \mid \cdots \mid l_n\, x_n \Rightarrow \varphi_n$	Propositional case
$\mid \lambda x{:}s.\ \varphi \mid \varphi\ e$	Predicates and application
$\mid e_1 = e_2$	Term equality
$\mid \forall x{:}s.\ \varphi \mid \exists x{:}s.\ \varphi \mid \exists! x{:}s.\ \varphi$	Term quantifiers

Sets

$s ::= \alpha \mid M.\alpha$	Set names
$\mid 1 \mid [x{:}s_1] \times s_2$	Unit and (dependent) cartesian product
$\mid 0 \mid l_1{:}s_1 + l_2{:}s_2$	Void and disjoint union
$\mid [x{:}s_1] \to s_2$	(Dependent) function space
$\mid \lambda x{:}s_1.\ s_2 \mid s\ e$	Dependent set and application
$\mid s/\rho$	Set quotient by an equivalence relation
$\mid \{x{:}s \mid \rho\}$	Subset satisfying a predicate
$\mid \texttt{rz}\ s$	Realizers of a set

Terms

$e ::= x \mid M.x$	Term names
$\mid \lambda x{:}s_1.\ e \mid e_1\ e_2$	Function and application
$\mid (e_1, \ldots, e_2) \mid \pi_n\, e$	Tuple and projection
$\mid l\, e \mid (\texttt{match}\, e_0\, \texttt{with}\, l_1\, x_1 \Rightarrow e_1 \mid l_2\, x_2 \Rightarrow e_2)$	Injection and projection from a union
$\mid [e]_\rho \mid \texttt{let}\ [x]_\rho = e_1\ \texttt{in}\ e_2$	Equivalence class and picking a representative
$\mid \texttt{rz}\ e \mid \texttt{let}\ \texttt{rz}\ x = e_1\ \texttt{in}\ e_2$	Realized value and picking a realizer
$\mid e : s$	Type coercion (e.g., in and out of a subset)
$\mid \iota\, x{:}s.\, \varphi$	Definite description
$\mid \texttt{let}\, x{=}e_1\, \texttt{in}\, e_2$	Local definition

Models

$M ::= m \mid M.m$	Model names
$\mid M_1\ M_2$	Application of parameterized model

Proposition Kinds

$\Pi ::= \texttt{Prop} \mid \texttt{Stable}$	Classifiers for all propositions/stable propositions
$\mid \texttt{Equiv}(s)$	Classifier for stable equivalences on s
$\mid [x{:}s] \to \Pi$	Classifier for a predicate/relation

Set Kinds

$\kappa ::= \texttt{Set}$	Classifier for a proper set
$\mid [x{:}s] \to \kappa$	Classifier for a dependent set

Theory Elements

$\theta ::= \texttt{Definition}\ x := e. \mid \texttt{Definition}\ \alpha := s.$	Give a name to a term or set
$\mid \texttt{Definition}\ p := \varphi. \mid \texttt{Definition}\ T := \Theta.$	Give a name to a predicate or theory
$\mid \texttt{Parameter}\ x : s. \mid \texttt{Parameter}\ \alpha : \kappa.$	Require an element in the given set or kind
$\mid \texttt{Parameter}\ p : \Pi. \mid \texttt{Parameter}\ m : \Theta.$	Require a predicate or model of the given sort
$\mid \texttt{Axiom}\ p : \varphi.$	Axiom that must hold

Theories

$\Theta ::= T$	Theory name
$\mid \texttt{thy}\ \theta_1, \ldots, \theta_n\ \texttt{end}$	Theory of a model
$\mid [m{:}\Theta_1] \to \Theta_2$	Theory of a uniform family of models
$\mid \lambda m{:}\Theta_1.\ \Theta_2 \mid \Theta\ m$	Parameterized theory and application

Fig. 3. Input Syntax (Simplified)

This says that the programmer should define a type s and a per $\approx_s$ on $[\![s]\!]$. Here $\approx_s$ is *not* an OCaml value of type $s \to s \to \texttt{bool}$, but an abstract relation on the set $[\![s]\!] \times [\![s]\!]$. The relation may be uncomputable.

The translation of the declaration of a dependent set $\texttt{Parameter t} : \texttt{s} \to \texttt{Set}$ uses uniform families (Section 2.2). The underlying type `t` is non-dependent, but the per $\approx_{\texttt{t}}$ receives an additional parameter $\texttt{x} : [\![\texttt{s}]\!]$.

A value declaration $\texttt{Parameter}\ x : s$ is translated to

```
val x : s
assertion x_support : x : ||s||
```

which requires the definition of a value `x` of type `s` which is in the support of `s`.

A value definition $\texttt{Definition x} := e$ where e is an expression denoting an element of `s` is translated to

```
val x : s
assertion x_def : x ≈s e
```

The assertion does *not* force `x` to be defined as e, only to be equivalent to it with respect to $\approx_{\texttt{s}}$. This is useful, as often the easiest way to define a value is not the most efficient way to compute it.

Constructions of sets in the input language are translated to corresponding constructions of modest sets. We comment on those that are least familiar.

Subsets. Given a predicate ϕ on a per A, the sub-per $\{x : A \mid \phi\}$ has underlying type $|A| \times |\phi|$ where $(u_1, v_1) \approx_{\{x:A|\phi\}} (u_2, v_2)$ when $u_1 \approx_A u_2$, $v_1 \Vdash \phi(u_1)$ and $v_2 \Vdash \phi(u_2)$. The point is that a realizer for an element of $\{x : A \mid \phi\}$ carries information about *why* the element belongs to the subset.

A type coercion $e : t$ can convert an element of the subset $s = \{x : t \mid \phi(x)\}$ to an element of t. At the level of realizers this is achieved by the first projection, which keeps a realizer for the element but forgets the one for $\phi(e)$. The opposite type coercion $e' : s$ takes an $e' \in t$ and converts it to an element of the subset. This is only well-formed when $\phi(e')$ is valid. Then, if $u \Vdash_t e'$ and $v \Vdash \phi(e')$, a realizer for $e' : s$ is (u, v). However, since RZ cannot in general know a v which validates $\phi(e')$, it emits the pair $(u, (\texttt{assure}\, v{:}|\phi|,\ \phi\, u\, v\, \texttt{in}\, v))$.

Quotients. Even though we may form quotients of pers by arbitrary equivalence relations, only quotients by $\neg\neg$-stable relations behave as expected.[7] A stable equivalence relation on a per A is the same thing as a partial equivalence relation ρ on $|A|$ which satisfies $\rho(x, y) \implies x \approx_A y$. Then the quotient A/ρ is the per with $|A/\rho| = |A|$ and $x \approx_{A/\rho} y \iff \rho(x, y)$.

Luckily, it seems that many equivalence relations occurring in computable mathematics are stable, or can be made stable with a little bit of manipulation. For example, the coincidence relation on Cauchy sequences is expressed by a $\forall\exists\forall$ formula, but if we restrict to the *rapid* Cauchy sequences, it becomes a

[7] The trouble is that from equality of equivalence classes $[x]_\rho = [y]_\rho$ we may conclude only $\neg\neg\rho(x, y)$ rather than the expected $\rho(x, y)$.

(negative) $\forall$ formula. It is interesting that most practical implementations of real numbers follow this line of reasoning and represent real numbers in way that avoids annotating every sequence with its rate of convergence.

Translation of an equivalence class $[e]_\rho$ is quite simple, since a realizer for e also realizes its equivalence class $[e]_\rho$. The elimination term `let` $[x]_\rho = \xi$ `in` e, means "let x be any element of ρ-equivalence class ξ in e". It is only well-formed when e does not depend on the choice of x, but this is something RZ cannot check. Therefore, if u realizes ξ, RZ uses u as a realizer for x and emits an obligation saying that the choice of a realizer for x does not affect e.

The underlying set of realizers. Another construction on a per A is the underlying per of realizers `rz` A, defined by $|\texttt{rz}\ A| = |A|$ and $u \approx_{\texttt{rz}\ A} vu \in \|A\| \wedge \iff u = v$, where by $u = v$ we mean observational equality of values u and v. An element $r \in \texttt{rz}\ A$ realizes a unique element $\texttt{rz}\ r \in A$. The elimination term `let rz` $x = e_1$ `in` e_2, which means "let x be any realizer for e_1 in e_2", is only well-formed if e_2 does not depend on the choice of x. This is an uncheckable condition, hence RZ emits a suitable obligation in the output, and uses for x the same realizer as for e_1.

The construction `rz` A validates the Presentation Axiom (see Section 7.3). In the input language it gives us access to realizers, which is useful because many constructions in computable mathematics, such as those in Type Two Effectivity [1], are explicitly expressed in terms of realizers.

5.2 Translation of propositions

The driving force behind the translation of logic is a theorem [16, 4.4.10] that says that under the realizability interpretation every formula ϕ is equivalent to one that says, informally speaking, "there exists $u \in |\phi|$, such that u realizes ϕ". Furthermore, the formula "u realizes ϕ" is computationally trivial. The translation of a predicate ϕ then consists of its underlying type $|\phi|$ and the relation $u \Vdash \phi$, expressed as a negative formula.

Thus an axiom `Axiom A` : ϕ in the input is translated to

```
val u : |ϕ|
assertion A : u ⊩ ϕ
```

which requires the programmer to validate ϕ by providing a realizer for it. When ϕ is a compound statement RZ computes the meaning as described in Figure 1.

In RZ we avoid the explicit realizer notation $u \Vdash \phi$ in order to make the output easier to read. A basic predicate declaration `Parameter` $p : s \to$`Prop` is translated to a type declaration `type ty_p` and a predicate declaration `predicate` p : $s \to$ `ty_p` $\to$ `bool` together with assertions that p is strict and extensional.

Frequently we know that a predicate is stable, which can be taken into account when computing its realizability interpretation. For this purpose the input language has the subkind `Stable` of `Prop`. When RZ encounters a predicate which is declared to be stable, such as $p : s \to$`Stable`, it does not generate a declaration of `ty_p` and it does not give p an extra argument.

Another special kind in RZ input language is the kind $\texttt{Equiv}(s)$ of stable equivalence relations on a set s. When an equivalence relation is declared with $\texttt{Parameter}\ p : \texttt{Equiv}(s)$, RZ will output assertions stating that p is strict, extensional, reflexive, symmetric and transitive.

6 Implementation

The RZ implementation consists of several sequential passes.

After the initial parsing, a *type reconstruction* phase checks that the input is well-typed (and checks for well-formedness to the extent that it is easily decidable), and if successful produces an annotated result with all variables explicitly tagged with types. The type checking phase uses a system of dependent types, with limited subtyping (implicit coercions) for sum types and subset types.

Next the realizability translation is performed as described in Section 5, producing interface code. The flexibility of the full input language (e.g., n-ary sum types and dependent product types) makes the translation code fairly involved, and so it is performed in a "naive" fashion whenever possible. The immediate result of the translation is not easily readable.

Thus, up to four more passes simplify the output before it is displayed to the user. A *thinning* pass removes all references to trivial realizers produced by stable formulas. An *optimization* pass applies an ad-hoc collection of basic logical and term simplifications in order to make the output more readable. Some redundancy may remain, but in practice the optimization pass helps significantly.

Finally, the user can specify two optional steps occur. RZ can perform a *phase-splitting* pass [17]. This is an experimental implementation of an transformation that can replace a functor (a relatively heavyweight language construct) by parameterized types and/or polymorphic values.

The other optional transformation is a *hoisting* pass which moves obligations in the output to top-level positions. Obligations appear in the output inside assertions, at the point where an uncheckable property was needed. Moving these obligations to the top-level make it easier to see exactly what one is obliged to verify, and can sometimes make them easier to read, at the cost of losing information about why the obligation was required at all.

7 Examples

In this section we look at several examples which demonstrate various points of RZ. Unfortunately, serious examples from computable mathematics take too much space[8] and will have to be presented separately. The main theme is that constructively reasonable axioms yield computationally reasonable operations.

[8] The most basic structure in analysis (the real numbers) alone requires several operations and a dozen or more axioms.

7.1 Decidable sets

A set S is said to be decidable when, for all $x, y \in S$, $x = y$ or $\neg(x = y)$. In classical mathematics all sets are decidable, but RZ requires an axiom

```
Parameter s : Set.
Axiom eq: ∀ x y : s, x = y ∨ ¬ (x = y).
```

to produce a realizer for equality

```
val eq : s → s → [‘or0 | ‘or1]
assertion eq : ∀ (x:‖s‖, y:‖s‖),
                   (match eq x y with
                      ‘or0 ⇒ x ≈s y
                    | ‘or1 ⇒ ¬ (x ≈s y) )
```

We read this as follows: `eq` is a function which takes arguments `x` and `y` of type `s` and returns ‘`or0` or ‘`or1`. If it returns ‘`or0`, then $\mathtt{x} \approx_{\mathtt{s}} \mathtt{y}$, and if it returns ‘`or1`, then $\neg(\mathtt{x} \approx_{\mathtt{s}} \mathtt{y})$. In other words `eq` is a decision procedure.

7.2 Inductive types

To demonstrate the use of dependent types we show how RZ handles general inductive types, also known as W-types or general trees [18]. Recall that a W-type is a set of well-founded trees, where the branching types of trees are described by a family of sets $B = \{T(x)\}_{x \in S}$. Each node in a tree has a *branching type* $x \in S$, which determines that the successors of the node are labeled by the elements of $T(x)$. Figure 4 shows an RZ axiomatization of W-types. The theory `Branching`

```
Parameter W : [B : Branching] →
thy
  Parameter w : Set.
  Parameter tree : [x : B.s] → (B.t x → w) → w.
  Axiom induction:
    ∀ M : thy Parameter p : w → Prop. end,
    (∀ x : B.s, ∀ f : B.t x → w,
       ((∀ y : B.t x, M.p (f y)) → M.p (tree x f))) →
    ∀ t : w, M.p t.
end.
```

Fig. 4. General inductive types

describes that a branching type consists of a set `s` and a set `t` depending on `s`. The theory `W` is parameterized by a branching type `B`. It specifies a set `w` of well-founded trees and a tree-forming operation `tree` with a dependent type $\Pi_{\mathtt{x} \in \mathtt{B.s}}(\mathtt{B.t(x)} \to \mathtt{w}) \to \mathtt{w}$. The inductive nature of `w` is expressed with the axiom `induction`, which states that for every property `M.p`, if `M.p` is an inductive property then every tree satisfies it. A property is said to be *inductive* if a tree `tree x f` satisfies it whenever all its successors satisfy it.

In the translation dependencies at the level of types and terms disappear. A branching type is determined by a pair of non-dependent types `s` and `t` but the per $\approx_{\mathtt{t}}$ depends on $[\![\mathtt{s}]\!]$. The theory `W` turns into a signature for a functor receiving a branching type `B` and returning a type `w`, and an operation `tree` of type $\mathtt{B.s} \to (\mathtt{B.t} \to \mathtt{w}) \to \mathtt{w}$. One can use phase-splitting to translate axiom `induction` into a specification of a polymorphic function

$$\mathtt{induction} : (\mathtt{B.s} \to (\mathtt{B.t} \to \mathtt{w}) \to (\mathtt{B.t} \to \alpha) \to \alpha) \to \mathtt{w} \to \alpha,$$

which is a form of recursion on well-founded trees. Instead of trying to explain what `induction` is supposed to do, we show a surprisingly simple, hand-written implementation of W-types in OCaml. The reader may enjoy figuring out how it works:

```
module W (B : Branching) = struct
  type w = Tree of B.s * (B.t -> w)
  let tree x y = Tree (x, y)
  let rec induction f (Tree (x, g)) =
    f x g (fun y -> induction f (g y))
end
```

7.3 Axiom of choice

RZ can help explain why a generally accepted axiom is not constructively valid. Consider the Axiom of Choice:

```
Parameter a b : Set.
Parameter r : a → b → Prop.
Axiom ac: (∀ x : a, ∃ y : b, r x y) →
          (∃ c : a → b, ∀ x : a, r x (c x)).
```

The relevant part of the output is

```
val ac : (a → b * ty_r) → (a → b) * (a → ty_r)
assertion ac :
  ∀ f:a → b * ty_r,
    (∀ (x:‖a‖), let (p,q) = f x in p : ‖b‖ ∧ r x p q) →
    let (g,h) = ac f in
      g : ‖a → b‖ ∧ (∀ (x:‖a‖), r x (g x) (h x))
```

This requires a function `ac` which accepts a function `f` and computes a pair of functions $(\mathtt{g}, \mathtt{h})$. The input function `f` takes an $\mathtt{x}{:}\|\mathsf{a}\|$ and returns a pair $(\mathtt{p}, \mathtt{q})$ such that `q` realizes the fact that `r x p` holds. The output functions `g` and `h` taking $\mathtt{x}{:}\|\mathsf{a}\|$ as input must be such that `h x` realizes `r x (g x)`. Crucially, the requirement $\mathtt{g}{:}\|\mathsf{a} \to \mathsf{b}\|$ says that `g` must be extensional, i.e., map equivalent realizers to equivalent realizers. We could define `h` as the first component of `f`, but we cannot hope to implement `g` in general because the second component of `f` is not assumed to be extensional.

The *Intensional* Axiom of Choice allows the choice function to depend on the realizers:

```
Axiom iac: (∀ x : a, ∃ y : b, r x y) →
           (∃ c : rz a → b, ∀ x : rz a, r (rz x) (c x)).
```

Now the output is

```
val iac : (a → b * ty_r) → (a → b) * (a → ty_r)
assertion iac :
  ∀ f:a → b * ty_r,
    (∀ (x:‖a‖),  let (p,q) = f x in p : ‖b‖ ∧ r x p q) →
    let (g,h) = iac f in
      (∀ x:a, x : ‖a‖ → g x : ‖b‖) ∧ (∀ (x:‖a‖),  r x (g x) (h x))
```

which is exactly the same as before, *except* that the troublesome requirement g:‖a → b‖ turned into ∀x:a. (x:‖a‖ ⇒ g x:‖b‖), which is weaker. We can implement `iac` in OCaml as

```
let iac f = (fun x -> fst (f x)), (fun x -> snd (f x))
```

The Intensional Axiom of Choice is in fact just an instance of the usual Axiom of Choice applied to `rz` A and B. Combined with the fact that `rz` A covers A, this establishes the validity of *Presentation Axiom* [19], which states that every set is an image of one satisfying the axiom of choice.

7.4 Modulus of Continuity

As a last example we show how certain constructive principles require the use of computational effects. To keep the example short, we presume that we are already given the set of natural numbers `nat` with the usual structure.

A *type 2 functional* is a map $f : (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat}$. It is said to be continuous if the output of $f(a)$ depends only on an initial segment of the sequence a. We can express the (non-classical) axiom that all type 2 functionals are continuous in RZ as follows:

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

The axiom says that for any `f` and `a` there exists $\mathtt{k} \in \mathtt{nat}$ such that $\mathtt{f}(\mathtt{b}) = \mathtt{f}(\mathtt{a})$ as soon as the sequences `a` and `b` agree on the first `k` terms. The axiom is translated to the specification

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

which says that `continuity f a` is a number `p` such that `f(a)` = `f(b)` whenever `a` and `b` agree on the first `p` terms. In other words, `continuity` is a *modulus of continuity* functional. It cannot be implemented in a purely functional language,[9] but with the use of store we can implement it in OCaml as

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

To compute a modulus for `f` at `a`, the program creates a function `a'` which is just like `a` except that it stores in `p` the largest argument at which it has been called. Then `f a'` is computed, its value is discarded, and the value of `p` is returned. The program works because `f` is assumed to be extensional and must therefore not distinguish between extensionally equal sequences `a` and `a'`.

8 Related Work

8.1 Coq

Coq provides complete support for theorem-proving and creating trusted code. A common pattern of use is to write code in Coq's functional language (values whose types are `Sets`), to state and prove theorems that the code behaves correctly (where the theorems are Coq values whose types are `Props`), and then have Coq extract correct code. In such cases, RZ is complementary to Coq; it can clarify the constructive content of mathematical structures and hence suggest an appropriate division between Coq's `Set` and `Prop`. (We hope RZ will soon be able to produce output in Coq syntax.)

8.2 Other tools

Komagata and Schmidt [8] describe a system that uses a similar realizability translation to ours. Like Coq, the system translates formal proofs to programs. An interesting technical difference is that the algorithm they use, attributed to John Hatcliff, does thinning as it goes along, rather than making a separate pass.

8.3 Other Models of Computability

Many formulations of computable mathematics are based on realizability models [13], even though they were not initially developed, (nor are they usually presented) within the framework of realizability: Recursive Mathematics [21] is based on the original realizability by Turing machines [22]; Type Two Effectivity [1] on function realizability [23] and relative function realizability [24], while topological and domain representations [25, 26] are based on realizability over the

[9] There are models of λ-calculus which validate the choice principle $AC_{2,0}$, but this contradicts the existence of a modulus of continuity functional, see [20, 9.6.10].

graph model $\mathcal{P}\omega$ [27]. A common feature is that they use models of computation which are well suited for the theoretical studies of computability.

Approaches based on simple programming languages augmented with datatypes for real numbers [28, 29] and topological algebras [2], and machines augmented with (suitably chosen subsets of) real numbers [30–32] are motivated by issues ranging from purely theoretical concerns about computability and complexity to practical questions in computational geometry. RZ attempts to improve practicality by using an actual real-world programming language, and by providing an input language which is rich enough to allow descriptions of involved mathematical structures that go well beyond the real numbers.

Finally, we hope that RZ and, hopefully, its forthcoming applications, give plenty of evidence for the *practical* value of Constructive Mathematics [33].

References

1. Weihrauch, K.: Computable Analysis. Springer, Berlin (2000)
2. Tucker, J., Zucker, J.I.: Computable functions and semicomputable sets on many-sorted algebras. In Abramsky, S., Gabbay, D., Maibaum, T., eds.: Handbook of Logic in Computer Science, Volume 5, Oxford, Clarendon Press (1998)
3. Blanck, J.: Domain representability of metric spaces. Annals of Pure and Applied Logic **83** (1997) 225–247
4. Edalat, A., Lieutier, A.: Domain of differentiable functions. In Blanck, J., Brattka, V., Hertling, P., Weihrauch, K., eds.: Computability and Complexity in Analysis. (2000) CCA2000 Workshop, Swansea, Wales, September 17–19, 2000.
5. Müller, N.: The iRRAM: Exact arithmetic in C++. In Blanck, J., Brattka, V., Hertling, P., Weihrauch, K., eds.: Computability and Complexity in Analysis. (2000) 319–350 CCA2000 Workshop, Swansea, Wales, September 17–19, 2000.
6. Lambov, B.: RealLib: an efficient implementation of exact real arithmetic. In Grubba, T., Hertling, P., Tsuiki, H., Weihrauch, K., eds.: Computability and Complexity in Analysis. (2005) 169–175 Proccedings, Second International Conference, CCA 2005, Kyoto, Japan, August 25–29, 2005.
7. Leroy, X., Doligez, D., Garrigue, J., Rémy, D., Vouillon, J.: The Objective Caml system, documentation and user's manual - release 3.08. Technical report, INRIA (July 2004)
8. Komagata, Y., Schmidt, D.A.: Implementation of intuitionistic type theory and realizability theory. Technical Report TR-CS-95-4, Kansas State University (1995)
9. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Springer (2004)
10. Benl, H., Berger, U., Schwichtenberg, H., Seisenberger, M., Zuber, W.: Proof theory at work: Program development in the Minlog system. In Bibel, W., Schmidt, P.H., eds.: Automated Deduction: A Basis for Applications. Volume II, Systems and Implementation Techniques. Kluwer Academic Publishers, Dordrecht (1998)
11. Longley, J.: Matching typed and untyped realizability. Electr. Notes Theor. Comput. Sci. **23**(1) (1999)
12. Longley, J.: When is a functional program not a functional program? In: International Conference on Functional Programming. (1999) 1–7
13. Bauer, A.: The Realizability Approach to Computable Analysis and Topology. PhD thesis, Carnegie Mellon University (2000)

14. Jacobs, B.: Categorical Logic and Type Theory. Elsevier Science (1999)
15. Post, E.: Recursive unsolvability of a problem of thue. The Journal of Symbolic Logic **12** (1947) 1–11
16. Troelstra, A.S., van Dalen, D.: Constructivism in Mathematics, An Introduction, Vol. 1. Number 121 in Studies in Logic and the Foundations of Mathematics. North-Holland (1988)
17. Harper, R., Mitchell, J.C., Moggi, E.: Higher-order Modules and the Phase Distinction. In: Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL '90). (1990) 341–354
18. Nordström, B., Petersson, K., Smith, J.M.: Programming in Martin-Löf's Type Theory. Oxford University Press (1990)
19. Barwise, J.: Admissible Sets and Structures. Springer-Verlag (1975)
20. Troelstra, A.S., van Dalen, D.: Constructivism in Mathematics, An Introduction, Vol. 2. Number 123 in Studies in Logic and the Foundations of Mathematics. North-Holland (1988)
21. Ershov, Y.L., Goncharov, S.S., Nerode, A., Remmel, J.B., eds.: Handbook of Recursive Mathematics. Elsevier, Amsterdam (1998)
22. Kleene, S.C.: On the interpretation of intuitionistic number theory. Journal of Symbolic Logic **10** (1945) 109–124
23. Kleene, S.C., Vesley, R.E.: The Foundations of Intuitionistic Mathematics, especially in relation to recursive functions. North-Holland Publishing Company (1965)
24. Birkedal, L.: Developing Theories of Types and Computability. PhD thesis, School of Computer Science, Carnegie Mellon University (December 1999)
25. Blanck, J.: Computability on topological spaces by effective domain representations. PhD thesis, Uppsala University, Department of Mathematics, Uppsala, Sweden (1997)
26. Bauer, A., Birkedal, L., Scott, D.S.: Equilogical spaces. Theoretical Computer Science **1**(315) (2004) 35–59
27. Scott, D.S.: Data types as lattices. SIAM Journal of Computing **5**(3) (1976) 522–587
28. Escardó, M.H.: PCF extended with real numbers. PhD thesis, Department of Computer Science, University of Edinburgh (December 1997)
29. Marcial-Romero, J.R., Escardó, M.H.: Semantics of a sequential language for exact real-number computation. In: Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science. (July 2004) 426–435
30. Borodin, A., Monro, J.I.: The computational complexity of algebraic and numeric problems. Number 1 in Elsevier computer science library : Theory of computation series. New York, London, Amsterdam : American Elsevier (1975)
31. Blum, L., Cucker, F., Shub, M., Smale, S.: Complexity and Real Computation. Springer-Verlag, New York (1998)
32. Yap, C.K.: Theory of real computation according to EGC (2006) To appear in LNCS Volume based on the Dagstuhl Seminar "Reliable Implementation of Real Number Algorithms: Theory and Practice", Jan 8-13, 2006.
33. Bishop, E., Bridges, D.: Constructive Analysis. Volume 279 of Grundlehren der math. Wissenschaften. Springer-Verlag (1985)