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 is not only a fundamental tool in logic and computability, but also has direct application to the design and implementation of programs: it can produce interfaces for the data structure corresponding to a mathematical theory. Our tool, called RZ, serves as a bridge between the worlds of constructive mathematics and programming. By using the realizability interpretation of constructive mathematics, RZ translates 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, the answer is obvious. A group would have a type whose values represent group elements, as well as a binary operation that is associative, a constant neutral element, and a unary inverse operator.

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.

The theory of realizability provides guidance in development of computable mathematics. Our work shows that realizability is not only a fundamental tool in logic and computability, but also has direct application to the design and implementation of programs: it can produce a description of the data structure (a code interface) directly corresponding to a mathematical specification.

However, doing this by hand quickly grows tedious. Worse, different but logically equivalent sets of axioms correspond to different, although interdefinable, interfaces for code; one might then want to compare several variations, since some interfaces will be more useful than others in practice. And few programmers — even those with strong backgrounds in mathematics and classical logic — are familiar with constructive logic or realizability. Programmers are used to language constructs describing interfaces (e.g., C++ header files, ML signatures, or Java interfaces) and logical assertions (e.g., preconditions and postconditions).

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).

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. At the other extreme, the output of RZ can be viewed as a possible *input* to a program extraction tool, where the the distinction between computational and non-computational parts (in Coq these are `Set` and `Prop`, respectively) has been automatically determined; a corresponding implementation would then be provided through theorem-proving and program extraction.

The paper is organized as follows. In Section 2 we present a version of realizability which is most suitable for our purposes. Sections 4 and 3 describe the input and the output language of RZ, while in Section 5 we explain how RZ translates from one to the other. Various implementation issues are discussed in Section 6, and examples of RZ at work are shown in Section 7. We conclude with remarks on related work in Section 8.

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

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

the set $\mathcal{G}$ of all finite simple[4] directed graphs with vertices labeled by distinct integers. An exemplar directed graph G is shown in Figure 1. A common repre-

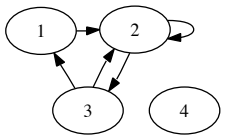

Fig. 1. A finite directed graph G

sentation 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. In our example, $\ell_V = [1; 2; 3; 4]$ and $\ell_A = [(1, 2); (2, 2); (2, 3); (3, 2); (3; 1)]$. Thus we define the datatype of graphs as[5]

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

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.,

1. The order in which vertices and arrows are listed is not important; for example, $[1; 2; 3; 4]$ and $[4; 1; 2; 3]$ represent the same vertices.
2. Each vertex and arrow must be listed exactly once.
3. 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, as long as they provide the usual ground types, product and function types.[6]

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

[4] At most one arrow between any two vertices.

[5] 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.

[6] It is also convenient to work with a language that supports sum types, as this allows a more natural representation of disjoint unions.

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 7.6. 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)*, which provides a theoretical basis for the definition of a realizability model that suits our needs.

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". There are two natural conditions that $\Vdash_{\mathcal{G}}$ ought to satisfy: (1) for every $G \in \mathcal{G}$ there should be at least one realizer (ℓ_V, ℓ_A) representing it, and (2) if (ℓ_V, ℓ_A) represents both G and G' then $G = G'$.[7] If (ℓ_V, ℓ_A) and (ℓ'_V, ℓ'_A) represent the same graph (e.g., because ℓ_V is a permutation of ℓ'_V, and similarly for ℓ_A and ℓ'_A) we say that they are *equivalent* and write $(\ell_V, \ell_A) \approx_{\mathcal{G}} (\ell'_V, \ell'_A)$. The relation $\approx_{\mathcal{G}}$ is a *partial* equivalence relation (symmetric and transitive, but not reflexive) because not every $(\ell_V, \ell_A) \in [\![\texttt{graph}]\!]$ represents a graph.

A general definition is in order. A *modest set* is 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.[8] 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.

[7] The latter condition is called *modesty* and is not strictly necessary for the development of the theory, though programmers would naturally expect it to hold.

[8] 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].

Modest sets form a category whose objects are modest sets and morphisms are the realized functions. A *realized function* $f : A \to B$ is a function $f : \langle A \rangle \to \langle B \rangle$ for which there exists $v \in [\![|A| \to |B|]\!]$ such that, for all $x \in \langle A \rangle$ and $u \in |A|$,

$$u \Vdash_A x \implies v\ u \Vdash_B f(x)\ . \tag{1}$$

This condition is just a mathematical expression of the usual idea that v is an implementation of f if it does to realizers what f does to the elements they represent.

The equivalent category of pers has as objects 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 $u, u' \in \|A\|$,

$$u \approx_A u' \implies v\ u \approx_B v\ u'\ . \tag{2}$$

Values v and v' that both satisfy (2) 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. For example, we may form a cartesian product $A \times B$ of pers A and B by

$$\begin{aligned} |A \times B| &= |A| * |B|, \\ (u_1, v_1) \approx_{A\times B} (u_2, v_2) &\iff u_1 \approx_A u_2 \wedge v_1 \approx_B v_2. \end{aligned}$$

The projections $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ are realized by `fst` and `snd`, respectively.

The morphisms between pers A and B again form a per B^A, also written as $A \to B$, called the *exponential* of A and B, with

$$\begin{aligned} |B^A| &= |A| \to |B|, \\ \|B^A\| &= \{v \in [\![|A| \to |B|]\!] \mid \forall\, u, u' \in [\![|A|]\!]\,.\, (u \approx_A u' \implies v\,u \approx_B v\,u')\} \\ u \approx_{B^A} v &\iff u, v \in \|A\| \wedge \forall\, w \in \|A\|\,.\, u\ w \approx_B v\ w. \end{aligned}$$

The evaluation map $e : B^A \times A \to B$ is realized by OCaml application, $\texttt{fun}\ (u, v) \to u\ v$. If a function $f : C \times A \to B$ is realized by v, then its transpose $\tilde{f} : C \to B^A$, $\tilde{f}(z)(x) = f(z, x)$, is realized by $\texttt{fun}\ z\ x \to v\ (z, x)$. This shows that the category of pers is cartesian closed. In Section 5.1 we review other canonical constructions on modest sets.

As an example we consider the cyclic group on seven elements $(\mathbb{Z}_7, 0, -, +)$. To implement the group, we must give a representation of $\mathbb{Z}_7$ as a per $Z = (|Z|, \approx_Z)$, and provide realizers for the neutral element 0, negation $-$, and addition $+$.

One possibility is to choose `int` as the underlying type $|Z|$, and to let $\|Z\|$ be only the integers `0` through `6`. Then negation and addition must work modulo `7` (i.e., must return an integer in the range `0`–`6` when given integers in this range). The neutral element would be the integer constant `0`, and the equivalence $\approx_Z$ would be integer equality.

Alternatively, we could take `int` as the underlying type $|Z|$, but let $\|Z\|$ include all integers. In this case, negation and addition could be simply integer addition and negation[9]. Here the neutral element could be implemented as any integer multiple of 7, and the equivalence $\approx_Z$ would be equivalence-modulo-7.

Both of these pers happen to be *decidable*, i.e., it can be algorithmically decided whether two values represent the same element of $\mathbb{Z}_7$, by code for integer equality and code for integer equivalence-modulo-7 respectively.

Not all pers are decidable. Examples include implementations of semigroups with an undecidable word problem [13], implementations of computable sets of integers (which might be realized by membership functions of type $\texttt{int} \to \texttt{bool}$), and implementations of computable real numbers (which might be realized by infinite Cauchy sequences). There is no presupposition that pers are computable (implementable). We can require decidable equivalence by adding a suitable axiom; see Section 7.1.

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 2. 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 modest set A may be viewed as a subset of $\langle A \rangle \times [\![|\phi|]\!]$, or a (possibly non-computable) function $\langle A \rangle \times [\![|\phi|]\!] \to \{\bot, \top\}$. In terms of pers, which is what RZ uses, 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 $\phi(u_2, v)$.

We illustrate how the realizability interpretation extracts the computational content of a proposition. To make an interesting example, suppose we have implemented the real numbers $\mathbb{R}$ as a per $R = (\texttt{real}, \approx_R)$, and consider the statement

[9] Taking care to prevent integer overflow.

Underlying types of realizers:

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

Realizers:

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

Fig. 2. Realizability interpretation of logic (outline)

that every cubic $x^3 + ax + b$ has a root,

$$\forall a{:}R.\ \forall b{:}R.\ \exists x{:}R.\ x^3 + ax + b = 0. \tag{3}$$

By computing according to Figure 2, we see that a realizer for this proposition is a value r of type $\mathtt{real} \to \mathtt{real} \to \mathtt{real} \times \mathtt{unit}$ 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 $\mathtt{real} \to \mathtt{real} \to \mathtt{real}$ 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[10] 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.

There are propositions whose realizers are "irrelevant" or free of computational content. For example, realizers for $\top$ and equality have type $\mathtt{unit}$. Another example is a negation $\neg\phi$, which is defined to be the same as $\phi \Rightarrow \bot$, whose realizers have type $|\phi| \to \mathtt{unit}$. Such realizers do not compute anything useful, and we may as well throw them away. Sometimes only a part of a realizer is computationally irrelevant, as we saw in the last example. 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. Any *negative* proposition, i.e., one built from $\top$, $\bot$, $=$, $\wedge$, $\Rightarrow$ and $\forall$ is

[10] 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.

stable, but there may be other propositions that are stable and are not written in the negative form.

It would be unproductive to bother the programmer with requirements for useless code. On input, one can specify whether abstract predicates have computational content. On output, extracted realizers go through a *thinning* phase, which removes irrelevant realizers.

2.3 Uniform families of modest sets

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 modest sets the appropriate notion is that of a *uniform* family $\{A_i\}_{i\in I}$, which is an assignment of a modest set $A_i = (\langle A_i\rangle, |A|, \Vdash_{A_i})$ to each $i \in \langle I\rangle$, where I is an index modest set [14, 6.3]. The uniformity comes from the requirement that all the A_i's share the same underlying type $|A_i| = |A|$. It is a desirable restriction from the implementation point of view, because it removes dependencies at the level of types. Note also that there is no dependency on the realizers, only on the elements of the underlying set.

We may express uniform families in terms of pers, too. 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 may form the *sum* $\Sigma_{i\in I} A_i$ of a uniform family $\{A_i\}_{i\in I}$ as

$$|\Sigma_{i\in I} A_i| = |I| \times |A|$$
$$(i_1, u_1) \approx_{\Sigma_{i\in I} A_i} (i_2, u_2) \iff i_1 \approx_I i_2 \wedge u_1 \approx_{A_{i_1}} u_2$$

and the *product* $\Pi_{i\in I} A_i$ as

$$|\Pi_{i\in I} A_i| = |I| \to |A|$$
$$\|\Pi_{i\in I} A_i\| = \{v \in [\![|I| \to |A|]\!] \mid \forall\, i, j \in [\![|I|]\!]\,.\,(i \approx_I j \implies v\, i \approx_{A_i} v\, j)\}$$
$$u \approx_{\Pi_{i\in I} A_i} v \iff u, v \in \|\Pi_{i\in I} A_i\| \wedge \forall\, i, j \in [\![|I|]\!]\,.\,(i \approx_I j \implies u\, i \approx_{A_i} v\, j)\,.$$

These constructions allow us to interpret (extensional) dependent type theory in the category of modest sets.

As an example of a uniform family we consider the cyclic group $(\mathbb{Z}_n, 0, -, +)$ of order n. To keep things simple, we assume that n ranges over natural numbers that can be represented by type `int` (i.e., $|N| = \texttt{int}$), and that $\approx_N$ is equality. The uniform family $\{Z_n\}_{n\in N}$ is then like the cyclic group of order 7, with 7

```
module type Ab =
sig
  type t
  val zero : t
  val neg : t -> t
  val add : t * t -> t
end
```

Fig. 3. The module type `Ab`.

replaced by n. Ignoring overflow, the underlying type would be $|Z_n| = \texttt{int}$. Any of the implementations suggested for $\mathbb{Z}_7$ would work here, with 7 replaced by the parameter n; in one case we would have $u \approx_{Z_n} v \iff u = v$ and in the other $u \approx_{Z_n} v \iff u \bmod n = v \bmod n$. Negation would be specified as a constant of dependent type $\Pi_{n \in N} Z_n \to Z_n$. Its realizer `neg` would then have type $|N| \to |Z_n| \to |Z_n|$, i.e., $\texttt{int} \to \texttt{int} \to \texttt{int}$, so that $\texttt{neg}(n)$ would be a realizer for negation on $\mathbb{Z}_n$. The realizer for addition would similarly take an extra argument n.

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.[11] 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. For example, if we look at the definition of module type `Ab` in Figure 3, we might guess that `Ab` is a signature for an Abelian group. However, `Ab` by itself requires an implementation satisfy only the signature of an Abelian group, but does not guarantee it behaves as an Abelian group. A complete description would contain the following further *assertions*:

1. there is a per $\approx_{\texttt{t}}$ on $[\![\texttt{t}]\!]$,
2. $\texttt{zero} \in \|\texttt{t}\|$.
3. for all $u, v \in [\![\texttt{t}]\!]$, if $u \approx_{\texttt{t}} v$ then $\texttt{neg}\ u \approx_{\texttt{t}} \texttt{neg}\ v$,
4. for all $u_1, u_2, v_1, v_2 \in [\![\texttt{t}]\!]$, if $u_1 \approx_{\texttt{t}} v_1$ and $u_2 \approx_{\texttt{t}} v_2$ then $\texttt{add}\ (u_1, u_2) \approx_{\texttt{t}} \texttt{add}\ (v_1, v_2)$,
5. for all $u \in \|\texttt{t}\|$, $\texttt{add}\ (\texttt{zero}, u) \approx_{\texttt{t}} u$,
6. for all $u \in \|\texttt{t}\|$, $\texttt{add}\ (u, \texttt{neg}\ u) \approx_{\texttt{t}} \texttt{zero}$,
7. for all $u, v, w \in \|\texttt{t}\|$, $\texttt{add}\ (\texttt{add}\ (u, v), w) \approx_{\texttt{t}} \texttt{add}\ (u, \texttt{add}\ (v, w))$,
8. for all $u, v \in \|\texttt{t}\|$, $\texttt{add}\ (u, v) \approx_{\texttt{t}} \texttt{add}(v, u)$.

[11] In object-oriented languages implementations and specifications are expressed with classes and interfaces, while in Haskell they correspond to modules and declarations.

Types

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

Terms

$e ::= x \mid M.x$	Term names
$\mid \mathtt{fun}\, x{:}\tau_1 \to e \mid e_1\, e_2$	Functions and application
$\mid () \mid (e_1, \ldots, e_n) \mid \mathtt{p}_n\, e$	Tuples and projection
$\mid \text{`}l\, e \mid (\mathtt{match}\ e\ \mathtt{with}\ \text{`}l_1\, x_1 \to e_1 \mid \cdots \mid \text{`}l_n\, x_n \to e_n)$	Injection and projection from a sum
$\mid \mathtt{assure}\, x{:}\tau,\, p\, \mathtt{in}\, e \mid \mathtt{assure}\, p\, \mathtt{in}\, e$	Obligations
$\mid \mathtt{let}\, x{=}e_1\, \mathtt{in}\, e_2$	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
$\mid \mathtt{match}\ e\ \mathtt{with}\ \text{`}l_1\, x_1 \Rightarrow p_1 \mid \cdots \mid \text{`}l_n\, p_n \Rightarrow e_n$	Propositional case
$\mid \mathtt{fun}\, x{:}\tau \to p \mid p\, e$	Propositional functions and application
$\mid e_1 \approx_s e_2 \mid e{:}\|s\|$	Pers and support
$\mid e_1{=}e_2$	(Observational) term equality
$\mid \forall x{:}\tau.\, p \mid \forall x{:}\|s\|.\, p$	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

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

Specification elements

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

Specifications (module types with assertions)

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

Fig. 4. The syntax of specifications (Simplified)

Assertions 2–4 state that `zero`, `neg`, and `add` realize a constant, a unary, and a binary operation, respectively, while assertions 5–8 correspond to axioms for Abelian groups.

The output of RZ consists of *module specifications*, each of which consists of a module type plus assertions about its components. More specifically, a 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.

A specification can describe an OCaml structure (a collection of definitions for types and values) or an OCaml functor (a parameterized module, i.e., a function mapping modules to modules). The latter would be appropriate, for example, when describing a uniform implementation of the real numbers that works given any implementation of natural numbers.

Assertions within module specifications (which appear as code comments) are expressed in the *negative fragment* of first-order logic, which contains constants for truth and falsehood, negation, conjunction, implication, equivalence, and universal quantification (but no disjunction or existential). For convenience we also introduce the propositional case

$$\texttt{match}\ e\ \texttt{with}\ \text{`}l_1\, x_1 \Rightarrow p_1 \mid \cdots \mid \text{`}l_n\, x_n \Rightarrow p_n$$

which is read "if e is of the form $\text{`}l_i\, x_i$ then p_i holds $(i = 1, \ldots, n)$", with the understanding that the expression is false if e does not match any case. This can be expressed with the negative formula

$$(\forall x_1.\, e = \text{`}l_1\, x_1 \Rightarrow p_1) \wedge \cdots \wedge (\forall x_n.\, e = \text{`}l_n\, x_n \Rightarrow p_n) \wedge \\ \neg((\forall x_1.\, e \neq \text{`}l_1\, x_1) \wedge \cdots \wedge (\forall x_n.\, e \neq \text{`}l_n\, x_n)).$$

The negative fragment is the part of first-order logic that has no computational content in the realizability interpretation. Consequently, the classical and constructive interpretations of assertions agree. This is quite desirable, since RZ acts as a bridge between constructive mathematics and real-world programmers, which typically are not familiar with constructive logic.

RZ ever produces only a small subset of OCaml types (the unit type, products, function types, polymorphic variant types, and parameter types). Correspondingly, the language of terms produced is fairly simple (tuples, functions, polymorphic variants, and local definitions). However, the programmer is free to implement a specification using any types and terms that exist in OCaml.

A special kind of term is an *obligation* $\texttt{assure}\, x{:}\tau,\, p\, \texttt{in}\, e$ 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 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 to be checked. The programmer must

replace each obligation with a value satisfying the obligation (i.e., demonstrate that the obligation can be satisfied). 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 simplest theory Θ is a list of *theory elements* $\theta_1 \ldots \theta_n$. A theory element may specify that a certain set, set element, proposition or predicate, or model must exist (using the Parameter keyword). It may also provide a definition of a set, term, proposition, predicate, or theory (using the Definition keyword). Finally, a theory element can be a named axiom (using the Axiom keyword).

We allow model parameters in theories; typical examples in mathematics include the theory of a vector space parameterized by a field of scalars, or the theory of the real numbers parameterized by a model of the natural numbers. Following Sannella, Sokolowski, and Tarlecki[12] [15] RZ supports two forms of parameterization. A theory of a parameterized implementation $[m{:}\Theta_1]{\to}\Theta_2$ describes a uniform family of models (i.e., a single implementation; a functor in OCaml) that maps every model m satisfying Θ_1 to a model of Θ_2. In contrast, a theory $\lambda m{:}\Theta_1.\ \Theta_2$ maps models to theories; if T is such a theory, then $T(M_1)$ and $T(M_2)$ are theories whose implementations might be completely unrelated.[13]

Propositions and predicates appearing in theories may use full first-order constructive logic, not just the negative fragment. The grammar for logical inputs is shown in Figure 4. Most of this should be familiar, including the use of lambda abstraction to define predicates.

The language of sets is rich, going well beyond the type systems of typical programming languages. In addition to any base sets postulated in a theory, one can construct dependent cartesian products and dependent function spaces. We also supports disjoint unions (with labeled tags), quotient spaces (a set modulo a stable equivalence relation), subsets (elements of a set satisfying a predicate). RZ even permits explicit references to sets of realizers.

The term language includes introduction and elimination constructs for the set level. For product sets we have tuples and projections ($\pi_1\, e$, $\pi_2\, e$, ...), 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

[12] "parameterized (program specification) $\neq$ (parameterized program) specification"
[13] Further, in some cases $T(M_1)$ might be implementable while $T(M_2)$ is not.

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
$\mid \texttt{rz}\ e \mid \texttt{let rz}\ x = e_1\ \texttt{in}\ e_2$	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. 5. Input Syntax (Simplified)

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 now explain how the translation from one to the other works. 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
```

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 $s \to s \to$ `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 `Parameter t : s → Set` follows the interpretation of dependent sets as uniform families (Section 2.3):

```
type t
predicate ≈t : s → t → t → bool
assertion strict_t :  ∀ x:s, y:t, z:t, y ≈t x z → x : ‖s‖
assertion extensional_t :
  ∀ x:s, y:s, z:t, w:t, x ≈s y → z ≈t x w → z ≈t y w
assertion symmetric_t : ∀ x:s, y:t, z:t, y ≈t x z → z ≈t x y
assertion transitive_t :
  ∀ x:s, y:t, z:t, w:t, y ≈t x z ∧ z ≈t x w → y ≈t x w
predicate ‖t‖ : s → t → bool
assertion support_def_t :  ∀ x:s, y:t, y : ‖t x‖ <→ y ≈t x y
```

The underlying output type `t` is still non-dependent, but the per is parameterized by `s`.

A value declaration `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`. When `s` is not a basic set, RZ computes the interpretation of the underlying type and support.

A value definition `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 $\approx_{\mathtt{s}}$. 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. In Section 2.1 we saw how products, exponentials and their dependent versions are formed. We briefly review the remaining constructions of modest sets. We only consider those constructions of terms that are not entirely straightforward.

Disjoint union. A disjoint union of modest sets $l_1{:}A + l_2{:}B$ is the modest set whose underlying type is the sum of underlying types,

$$|l_1{:}A + l_2{:}B| = \text{`}l_1 \,\mathtt{of}\, |A| + \text{`}l_2 \,\mathtt{of}\, |B|,$$

and the per is the disjoint union of pers $\approx_A$ and $\approx_B$, so that we have

$$\begin{aligned} \text{`}l_1\, u \approx_{l_1:A+l_2:B} \text{`}l_1\, v &\iff u \approx_{s_1} v, \\ \text{`}l_2\, u \approx_{l_1:A+l_2:B} \text{`}l_2\, v &\iff u \approx_{s_2} v. \end{aligned}$$

Subsets. The construction of subsets may look surprising at first, but it makes sense computationally. 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 $(u, (\mathtt{assure}\, v{:}|\phi|,\, \phi\, u\, v\, \mathtt{in}\, v))$.

Quotients. The category of modest sets has coequalizers, hence a quotient modest set A/ρ may be constructed for an any equivalence relation ρ on A. However, because equality does not carry any computational content, equality of equivalence classes $[x]_\rho = [y]_\rho$ implies only $\neg\neg\rho(x, y)$, not the usual $\rho(x, y)$. As this may cause confusion and mistakes, it is better to permit only quotients by *stable* equivalence relations, which behave as expected.

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, Cauchy sequences $(a_n)_{n\in\mathbb{N}}$ and $(b_n)_{n\in\mathbb{N}}$ represent the same real number when

$$\forall\, i \in \mathbb{N}\,.\, \exists\, j \in \mathbb{N}\,.\, \forall m, n \geq j.\, |a_m - b_n| \leq 2^{-i}\,. \tag{4}$$

This defines an equivalence relation on the set of Cauchy sequences which does not seem to be stable; intuitively a realizer for this equivalence would be a computation telling us at what point in the sequence the terms will be within 2^{-i} of each other. However, if we restrict attention just to the *rapid* Cauchy sequences, i.e., those satisfying $\forall\, i \in \mathbb{N}\,.\, |a_{i+1} - a_i| \leq 2^{-i}$, then the equivalence relation becomes

$$\forall\, i \in \mathbb{N}\,.\, |a_i - b_i| \leq 2^{-i+3},$$

which is a negative formula; the above realizer is rendered unnecessary. 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 $\texttt{let}\ [x]_\rho = \xi\ \texttt{in}\ e$, 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 $\texttt{rz}\ A$, defined by

$$\begin{aligned} |\texttt{rz}\ A| &= |A| \\ u \approx_{\texttt{rz}\ A} v &\iff u \in \|A\| \wedge u = v, \end{aligned}$$

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 $\texttt{let rz}\ x = e_1\ \texttt{in}\ e_2$, 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 $\texttt{rz}\ A$ validates the Presentation Axiom (see Section 7.5). 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.

Definite description. Russell's definite description operator $\iota\, x{:}s.\, \phi(x)$ denotes the unique element of $\{x{:}s \mid \phi(x)\}$. In case such an x does not exist, or if there are several, the term is not well formed. The RZ translation essentially just asks the programmer to provide suitable realizers for x and for $\phi(x)$, and to check uniqueness,

$$\texttt{assure}\, x{:}s, b{:}|\phi|,\ (x{:}\|s\| \wedge \phi\, x\, b \wedge \forall x'{:}s.\, \forall c{:}|\phi|.\, (\phi\, x'\, c \Rightarrow x \approx_s x'))\ \texttt{in}\, (x, b).$$

This is the best RZ can do, since in general it can check neither that x exists, nor that it is unique.

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 of $u \Vdash \phi$ as described in Figure 2.

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

```
type ty_p
predicate p : s → ty_p → bool
assertion strict_p :  ∀ x:s, a:ty_p, p x a → x : ‖s‖
assertion extensional_p :
   ∀ x:s, y:s, a:ty_p, x ≈ₛ y → p x a → p y a
```

We see that the predicate p has gained an additional argument of type `ty_p` (which the programmer is supposed to define in an implementation), and we write $p\, x\, a$ instead of $a \Vdash p\, x$. The two assertions require that p be strict and extensional with respect to $\approx_s$.

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 `Equiv`(s) of stable equivalence relations on a set s. When an equivalence relation is declared with `Parameter` p : `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. The details are fairly standard, so are omitted here. One non-obvious consequence of the realizability translation, however, is that the subset types with provably-equal predicates, e.g., $\{x{:}\alpha \mid \rho_1(x) \wedge \rho_2(x)\}$ and $\{x{:}\alpha \mid \rho_2(x) \wedge \rho_1(x)\}$ are isomorphic but not equal in general. An explicit coercion is required to go from one type to the other, because subset values will be pairs containing realizers for $\rho_1(x) \wedge \rho_2(x)$ and $\rho_2(x) \wedge \rho_1(x)$, and these realizers have potentially different types $|\rho_1(x)| * |\rho_2(x)|$ and $|\rho_2(x)| * |\rho_1(x)|$ respectively.

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. For example, direct translation of the `free` axiom in the output for Kuratowski-finite sets, see Figure 7 and Section 7.3, yields a value specification for `free` of type

$$(\texttt{A.a} \to \texttt{S.s}) \to (\texttt{fin} \to \texttt{S.s}) * (\texttt{unit} * (\texttt{A.a} \to \texttt{unit}) * (\texttt{fin} \to \texttt{fin} \to \texttt{unit}))$$

where `unit` is the unit (terminal) type classifying the trivial realizer. Thinning replaces this by the isomorphic type

$$(\texttt{A.a} \to \texttt{S.s}) \to \texttt{fin} \to \texttt{S.s}$$

and appropriately modifies references to `free` in the assertions to account for this change in type.

An *optimization* pass applies an ad-hoc collection of basic logical and term simplifications in order to make the output more readable. Logical simplifications include applications of truth table rules ($\top \wedge \varphi$ becomes φ), detection of syntactically identical premises and conclusions ($\varphi_1 \Rightarrow \varphi_1 \wedge \varphi_2$ becomes $\varphi_1 \Rightarrow \varphi_2$), and optimization of other common patterns we have seen arise ($\forall x{:}s.\ (x = e) \Rightarrow \rho(x)$ becomes $\rho(e)$). 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 idea is that although functors map modules containing types and terms to other modules containing

types and terms, constraints on the programming language ensure that output types depend only on input types (and not input terms). Thus, we can split each functor into a mapping from input types to output types, and then a separate (polymorphic) term mapping input types and terms to an output term. See Section 7.3 for an example.

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. See Section 7.2 for an example of hoisting.

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[14] and will have to be presented separately. The main theme is that constructively reasonable axioms yield computationally reasonable operations.

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, because decidability of equality is just an instance of the law of excluded middle. 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 which tells when values `x` and `y` represent the same element of the modest set.

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

7.2 Examples with obligations

In this section we show how RZ produces obligations, is sometimes able to optimize them away, and show the effect of hoisting.

Consider how we might define division of real numbers. Assuming the set of real numbers `real`, a constant `zero`, and multiplication operation $*$ have already been declared and axiomatized, we might write:

```
Definition nonZeroReal := {x : real | ¬ (x = zero)}.
Parameter inv : nonZeroReal → real.
Axiom inverse : ∀ x : real, ¬ (x = zero) -> x * (inv x) = one.
Definition (/) (x : real) (y : nonZeroReal) := x * (inv y).
```

We have defined the set of non-zero reals `nonZeroReal` and the inverse operation `inv` on it. Division `x/y` is defined as `x` $*$ `inv y`. This does *not* mean that the programmer must necessarily implement division this way, only that the implementation of `x/y` must be equivalent to `x` $*$ `inv y`.

In the axiom `inverse`, RZ encounters the subexpression `inv x`. Because `x` is quantified as an element of `real` rather than `nonZeroReal`, the typechecking phase inserts a coercion that makes the expression well-typed. Translation sees `inv(x : nonZeroReal)` instead of `inv x` and translates this to

```
inv (assure u:unit, ¬ (x ≈real zero) in (x, u))
```

If this were the final output, the programmer would have to verify that x is not zero, and provide a trivial realizer for it. However, in this case the thinning phase first removes the trivial realizer,

```
inv (assure ¬ (x ≈real zero) in x)
```

and then the optimizer determines that the obligation is not needed because the whole expressions appears under the hypothesis that x is not zero. So in the end the programmer sees

```
assertion inverse :
  ∀ (x:‖real‖),  ¬ (x ≈real zero) → (x * inv x) ≈real one
```

Assuming further that a strict linear order $<$ on `real` has been axiomatized, we might proceed by relating it to `inv`:

```
Axiom inv_positive: ∀ x : real, zero < x → zero < inv x.
```

Once again `inv x` appears in the input, but this time the optimizer is unable to remove the obligation, so the output is

```
assertion inv_positive: ∀ (x:‖real‖),
   zero < x → zero < inv (assure (not (x ≈real zero)) in x)
```

Local obligations can sometimes be hard to read, but if we activate the hoisting phase (see Section 6), the obligation can be moved to the top level. As this is done, the hypotheses under which the obligation appears are collected, and we get

```
assertion inv_positive:
  assure (∀ (x:‖real‖),  zero < x → not (x ≈real zero))
    in ∀ (x:‖real‖),  zero < x → zero < inv x
```

Now it is easier to understand what must be checked, namely that positive reals are not zero—an easy consequence of irreflexivity of $<$, but not something that RZ optimizer is aware of.

Lastly, we could define the golden ratio as the positive solution of $x^2 = x + 1$,

```
the x : real, (zero < x ∧ x*x = x + one)
```

Not surprisingly, RZ cannot determine that there is a unique such x, so it outputs an obligation:

```
assure x:real,
  (x : ‖real‖ ∧ zero < x ∧ x * x =real= x + one ∧
    (∀ (x':‖real‖),  zero < x' ∧ x' * x' ≈real x' + one → x ≈real x'))
  in x
```

7.3 Finite sets

There are many characterizations of finite sets, but the one that works best constructively is due to Kuratowski, who identified the finite subsets of A as the least family $K(A)$ of subsets of A that contains the empty set and is closed under unions with singletons. This characterization relies on powersets, which are not available in RZ. But the gist of it, namely that $K(A)$ is an *initial* structure a suitable sort, can be expressed as follows.

Recall that a $\vee$-*semilattice* is a set S with a constant $0 \in S$ and an associative, commutative, and idempotent operation "join" $\vee$ on S such that 0 is the neutral element for $\vee$, see Figure 6 for RZ axiomatization of semilattices. The Kuratowski finite sets $K(A)$ are the *free* semilattice generated by a set A, where $\vee$ is union and 0 is the empty set. This is formalized in RZ as shown in Figure 7. The theory `K` is parametrized by a model `A` which contains a set `a`. In the first line we

```
Definition Semilattice :=
thy
  Parameter s : Set.
  Parameter zero : s.
  Parameter join : s → s → s.
  Implicit Type x y z : s.
  Axiom commutative: ∀ x y,   join x y = join y x.
  Axiom associative: ∀ x y z, join (join x y) z = join x (join y z).
  Axiom idempotent:  ∀ x,     join x x = x.
  Axiom neutral:     ∀ x,     join x zero = x.
end.
```

Fig. 6. The theory of a semilattice

```
Definition K (A : thy
                    Parameter a : Set.
                  end) :=
thy
  include Semilattice.
  Parameter singleton : A.a → s.
  Definition fin := s.
  Definition emptyset := zero.
  Definition union := join.

  Axiom free :
    ∀ S : Semilattice, ∀ f : A.a → S.s,
    ∃! g : fin → S.s,
      g emptyset = S.zero ∧
        (∀ x : A.a, f x = g (singleton x)) ∧
        (∀ u v : fin, g (union u v) = S.join (g u) (g v)).
end.
```

Fig. 7. Kuratowski finite sets

include the theory of semilattices. Then we postulate an operation `singleton` which injects the generators into the semilattice. The three definitions are just a convenience, so that we can refer to the parts of `K(A)` by their natural names, e.g., `emptyset` instead of `zero`. The axiom `free` expresses the fact that `K(A)` is the free semilattice on `A.a`: for every semilattice `S` and a map `f` : `A.a` → `S.s` from the generators to the underlying set of `S`, there exists a unique semilattice homomorphism `g` : `fin` → `S.s` such that `f(x)` = `g(singleton x)`.

The output for `Semilattice` and `K` specifies values of suitable types for each declared constant and operation. All axioms but the last one are equations and have straightforward translations in terms of underlying pers. The output for the axiom `free` is shown in Figure 8. Because the axiom quantifies over all models `S` of the theory `Semilattice` its translation is a functor `Free` which accepts an implementation of a semilattice S and yields a realizer `free` validating the axiom. The computational meaning of `free` is a combination map and fold operation, taking a map `f` : `A.a` → `S.s` and a finite set `u` = $\{x_1, \ldots, x_n\}$, and return `f`$(x_1) \vee \cdots \vee$ `f`(x_n), where $\vee$ is the join operation on the semilattice S.

Applying phase-splitting to this axiom yields the even simpler specification

$$\texttt{val free} : \alpha \to (\alpha \to \alpha \to \alpha) \to (\texttt{A.a} \to \alpha) \to \texttt{fin} \to \alpha$$

(with an appropriate assertion) which replaces the module parameter `S` by two extra term arguments term (corresponding to the module components `S.zero` and `S.join`) and a type argument α for the type of lattice elements (corresponding to the module input `S.s`). This is even more recognizable as a folding operation over the set.

It is important to note that, in contrast to `fold` operators found in typical functional languages, `free` is only expected to work for suitable `join` arguments

```
module Free : functor (S : Semilattice) →
sig
  val free : (A.a → S.s) → fin → S.s
  assertion free :
    ∀ (f:‖A.a → S.s‖), let g = free f in
      g : ‖fin → S.s‖ ∧ g emptyset ≈S.s S.zero ∧
      (∀ (x:‖A.a‖),  f x ≈S.s g (singleton x)) ∧
      (∀ (u:‖fin‖, v:‖fin‖), g (union u v) ≈S.s S.join (g u) (g v)) ∧
      (∀ h:fin → S.s,  h : ‖fin → S.s‖ ∧ h emptyset ≈S.s S.zero ∧
         (∀ (x:‖A.a‖), f x ≈S.s h (singleton x)) ∧
         (∀ (u:‖fin‖, v:‖fin‖), h (union u v) ≈S.s S.join (h u) (h v)) →
         ∀ x:fin, y:fin,  x ≈fin y → g x ≈S.s h y)
end
```

Fig. 8. Output of axiom `free`.

(e.g., idempotent and order independent). These sets are not the typical finite-set data structure: there is no membership predicate, nor is there a way to compute the size of a set. There is no assumption that equality is decidable for set elements; this permits finite sets of exact real numbers, for example. Decidable equality is required both for membership and for detecting whether the same element has been added twice to the same set[15].

Some operations are nevertheless computable. Using `free` one can determine whether a finite set is empty. In the case of a set of exact real numbers, we cannot compute their sum, but we could compute maximum or minimum.

More common set implementations (e.g., the `Set` module in the OCaml standard library) implement sets over values with decidable total order; these could also be formalized in RZ.

7.4 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 $T(x)$. For example, to get non-empty binary trees whose leaves are labeled by natural numbers, define

$$\begin{aligned} S &= \{\texttt{cons}\} \cup \{\texttt{leaf}(n) \mid n \in \mathbb{N}\} \\ T(\texttt{cons}) &= \{\texttt{left}, \texttt{right}\} \\ T(\texttt{leaf}(n)) &= \emptyset. \end{aligned}$$

[15] The natural implementation would thus be an unordered collection of elements, possibly with duplicates.

Then a node of type `cons` has two successors, indexed by constants `left` and `right`, while a node of type `leaf`(n) does not have any successors.

Figure 9 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. 9. 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 $\Pi_{\mathtt{x}\in\mathtt{B.s}}(\mathtt{B.t(x)} \to \mathtt{w}) \to \mathtt{w}$. Given a branching type `x` and a map `f` : $\mathtt{B.t(x)} \to \mathtt{w}$, `tree x f` is the tree whose root has branching type `x` and whose successor labeled by $\ell \in \mathtt{B.t(x)}$ is the tree $\mathtt{f}(\ell)$. 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, see Appendix A for a complete output, 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
```

7.5 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 $\mathtt{x}{:}\|\mathtt{a}\|$ and returns a pair $(\mathtt{p}, \mathtt{q})$ such that `q` realizes the fact that $\mathtt{r}\ \mathtt{x}\ \mathtt{p}$ holds. The output functions `g` and `h` taking $\mathtt{x}{:}\|\mathtt{a}\|$ as input must be such that $\mathtt{h}\ \mathtt{x}$ realizes $\mathtt{r}\ \mathtt{x}\ (\mathtt{g}\ \mathtt{x})$. Crucially, the requirement $\mathtt{g}{:}\|\mathtt{a} \to \mathtt{b}\|$ 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 $\mathtt{g}{:}\|\mathtt{a} \to \mathtt{b}\|$ turned into $\forall \mathtt{x}{:}\mathtt{a}.\, (\mathtt{x}{:}\|\mathtt{a}\| \Rightarrow \mathtt{g}\ \mathtt{x}{:}\|\mathtt{b}\|)$, 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 $\mathtt{rz}\ A$ and B. Combined with the fact that $\mathtt{rz}\ A$ 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.

7.6 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 $\mathtt{f}(\mathtt{a}) = \mathtt{f}(\mathtt{b})$ 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,[16] 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 $\mathtt{a}'$ which is just like `a` except that it stores in `p` the largest argument at which it has been called. Then $\mathtt{f}\ \mathtt{a}'$ 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 $\mathtt{a}'$.

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

[16] 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].

correctly (where the theorems are Coq values whose types are `Prop`s), 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.)

In general, RZ is a smaller and more lightweight system and thus more flexible where it applies. It is not always practical or necessary to do theorem proving in order to provide an implementation; interfaces generated by RZ can be implemented in any manner. And, RZ provides a way to talk with programmers about constructive mathematics without bringing in full theorem proving.

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. For example, the translation of the conjunction-introduction rule has four cases, depending on whether the left and/or right propositions being proved are almost negative, in which case the trivial contribution can be immediately discarded.

8.3 Other Models of Computability

Many formulations of computable mathematics are based on realizability models [21], even though they were not initially developed, (nor are they usually presented) within the framework of realizability: Recursive Mathematics [22] is based on the original realizability by Turing machines [23]; Type Two Effectivity [1] on function realizability [24] and relative function realizability [25], while topological and domain representations [26, 27] are based on realizability over the graph model $\mathcal{P}\omega$ [28]. A common feature is that they use models of computation which are well suited for the theoretical studies of computability.

Other approaches are based on simple programming languages augmented with datatypes for real numbers [29, 30] and topological algebras [2], or machine models augmented with (suitably chosen subsets of) real numbers such as Real RAM [31], the Blum-Smale-Shub model [32], and the Exact Geometric Computation model [33]. The motivation behind these ranges from purely theoretical concerns about computability and complexity to practical issues in the design of programming languages and algorithms 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 [34].

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. Post, E.: Recursive unsolvability of a problem of thue. The Journal of Symbolic Logic **12** (1947) 1–11
14. Jacobs, B.: Categorical Logic and Type Theory. Elsevier Science (1999)
15. Sannella, D., Sokolowski, S., Tarlecki, A.: Toward formal development of programs from algebraic specifications: parameterisation revisited. Acta Informatica **29**(9) (1992) 689–736
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. Bauer, A.: The Realizability Approach to Computable Analysis and Topology. PhD thesis, Carnegie Mellon University (2000)
22. Ershov, Y.L., Goncharov, S.S., Nerode, A., Remmel, J.B., eds.: Handbook of Recursive Mathematics. Elsevier, Amsterdam (1998)
23. Kleene, S.C.: On the interpretation of intuitionistic number theory. Journal of Symbolic Logic **10** (1945) 109–124
24. Kleene, S.C., Vesley, R.E.: The Foundations of Intuitionistic Mathematics, especially in relation to recursive functions. North-Holland Publishing Company (1965)
25. Birkedal, L.: Developing Theories of Types and Computability. PhD thesis, School of Computer Science, Carnegie Mellon University (December 1999)
26. Blanck, J.: Computability on topological spaces by effective domain representations. PhD thesis, Uppsala University, Department of Mathematics, Uppsala, Sweden (1997)
27. Bauer, A., Birkedal, L., Scott, D.S.: Equilogical spaces. Theoretical Computer Science **1**(315) (2004) 35–59
28. Scott, D.S.: Data types as lattices. SIAM Journal of Computing **5**(3) (1976) 522–587
29. Escardó, M.H.: PCF extended with real numbers. PhD thesis, Department of Computer Science, University of Edinburgh (December 1997)
30. 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
31. 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)
32. Blum, L., Cucker, F., Shub, M., Smale, S.: Complexity and Real Computation. Springer-Verlag, New York (1998)
33. 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.
34. Bishop, E., Bridges, D.: Constructive Analysis. Volume 279 of Grundlehren der math. Wissenschaften. Springer-Verlag (1985)

A A complete example

To give at least one complete example, we include here an unabridged output for the theory of inductive types shown in Figure 9.

```
module type Branching =
 sig
   type s

   (** predicate (=s=) : s -> s -> bool *)
   (**  assertion symmetric_s :  forall x:s, y:s,  x =s= y -> y =s= x

        assertion transitive_s :
          forall x:s, y:s, z:s,  x =s= y /\ y =s= z -> x =s= z
   *)
```

```
  (** predicate ||s|| : s -> bool *)
  (**  assertion total_def_s :  forall x:s,  x : ||s|| <-> x =s= x
  *)

  (** branching types *)

  type t

  (** predicate (=t=) : s -> t -> t -> bool *)
  (**  assertion strict_t :  forall x:s, y:t, z:t,  y =(t x)= z -> x : ||s||

       assertion extensional_t :
         forall x:s, y:s, z:t, w:t,  x =s= y -> z =(t x)= w -> z =(t y)= w

       assertion symmetric_t :
         forall x:s, y:t, z:t,  y =(t x)= z -> z =(t x)= y

       assertion transitive_t :
         forall x:s, y:t, z:t, w:t,  y =(t x)= z /\ z =(t x)= w ->
           y =(t x)= w
  *)

  (** predicate ||t|| : s -> t -> bool *)
  (**  assertion total_def_t :
         forall x:s, y:t,  y : ||t x|| <-> y =(t x)= y
  *)

  (** branch labels *)
 end

module W : functor (B : Branching) ->
sig
  type w

  (** predicate (=w=) : w -> w -> bool *)
  (**  assertion symmetric_w :
         forall x:w, y:w,  x =w= y -> y =w= x

       assertion transitive_w :
         forall x:w, y:w, z:w,  x =w= y /\ y =w= z -> x =w= z
  *)

  (** predicate ||w|| : w -> bool *)
  (**  assertion total_def_w :  forall x:w,  x : ||w|| <-> x =w= x
  *)

  val tree : B.s -> (B.t -> w) -> w
  (**  assertion tree_support :
         forall x:B.s, y:B.s,  x =B.s= y ->
```

```
          forall f:B.t -> w, g:B.t -> w,
            (forall z:B.t, t:B.t,  z =(B.t x)= t -> f z =w= g t) ->
            tree x f =w= tree y g
  *)

  val induction : (B.s -> (B.t -> w) -> (B.t -> 'ty_p) -> 'ty_p) -> w -> 'ty_p
  (**  assertion 'ty_p [p:w -> 'ty_p -> bool] induction :
         (forall x:w, a:'ty_p,  p x a -> x : ||w||) ->
         (forall x:w, y:w, a:'ty_p,  x =w= y -> p x a -> p y a) ->
         forall f:B.s -> (B.t -> w) -> (B.t -> 'ty_p) -> 'ty_p,
           (forall (x:||B.s||),
              forall f':B.t -> w,
                (forall y:B.t, z:B.t,  y =(B.t x)= z ->
                   f' y =w= f' z) ->
                forall g:B.t -> 'ty_p,
                  (forall y:B.t,  y : ||B.t x|| -> p (f' y) (g y)) ->
                  p (tree x f') (f x f' g)) ->
           forall (t:||w||),  p t (induction f t)
  *)
end
```