Math. Struct. in Comp. Science (2011), *vol.* 21, *pp.* 861–882. © Cambridge University Press 2011
doi:10.1017/S0960129511000132

Classical mathematics for a constructive world

RUSSELL O'CONNOR

Department of Computing and Software, McMaster University,
Hamilton, Ontario, Canada
and
Microsoft Research – INRIA Joint Centre
Email: roconn@mcmaster.ca

Received 11 July 2010; revised 15 December 2010

Interactive theorem provers based on dependent type theory have the flexibility to support both constructive and classical reasoning. Constructive reasoning is supported natively by dependent type theory, and classical reasoning is typically supported by adding additional non-constructive axioms. However, there is another perspective that views constructive logic as an extension of classical logic. This paper will illustrate how classical reasoning can be supported in a practical manner inside dependent type theory without additional axioms. We will show several examples of how classical results can be applied to constructive mathematics. Finally, we will show how to extend this perspective from logic to mathematics by representing classical function spaces using a weak value monad.

1. Introduction

The common view of constructive logic is that it is a restriction of classical logic where the law of the excluded middle and proof by contradiction are not allowed. Several software proof assistants are founded on constructive logic, including Coq (Coq Development Team 2009) and Agda (Norell 2007), and use the Curry–Howard isomorphism to interpret constructive deductions as programs and *vice versa*.

In accordance with this common view of constructive logic, these systems can be extended to classical logic by adding non-constructive axioms such as the law of excluded middle or proof by contradiction. When these axioms are added, all the constructive theorems can be used in classical proofs, since they are all still valid classical deductions. However, one cannot use classical proofs in constructive theorems since they rely on non-constructive axioms.

There is a high price to be paid for these non-constructive axioms. If one attempts to evaluate a classical deduction (viewed through the Curry–Howard isomorphism), the evaluation could get stuck at a non-constructive axiom because these axioms do not have a computational interpretation.

We will not be taking this common viewpoint in this paper. Instead, we will see constructive logic as an extension of classical logic. We will add two new logical connectives to classical logic: the constructive disjunction, written $\varphi + \psi$, and the constructive existential, written $\Sigma a : A.\varphi(a)$. Constructive theorems will make use of these new connectives, while theorems living in the classical fragment will not. These constructive connectives are stronger than their classical counterparts in the sense that

https://doi.org/10.1017/S0960129511000132 Published online by Cambridge University Press

the following formulas hold:

$$(\varphi + \psi) \Rightarrow (\varphi \vee \psi) \tag{1}$$

$$(\Sigma a : A.\varphi(a)) \Rightarrow (\exists a : A.\varphi(a)). \tag{2}$$

This viewpoint allows us to mix classical and constructive theorems freely since a classical theorem is simply a special type of constructive theorem. In many cases, constructive results can be used in classical theorems by weakening them using the above theorems, but we will also see some examples where classical theorems are used in constructive proofs.

This alternative viewpoint is not original, but it is perhaps not widely known in the classical mathematics community. Even in constructive circles where it is known to some people, it is perhaps underappreciated. The purpose of this paper is to:

(1) Show that this alternative viewpoint is a useful way to practice classical and constructive mathematics.
(2) Show that classical theorems, when seen this way, are useful when creating constructive proofs.
(3) Extend this viewpoint from logic to mathematics by defining a classical function space as a special kind of constructive function space.

Before proceeding, it is worth remarking on my notation. Usually papers on constructive mathematics usurp the classical logical symbols, taking $\vee$ and $\exists$ to mean constructive disjunction and constructive existence, respectively. I feel that this obscures the fact that the constructive result is stronger than the classical result. Also, taking the same symbols to have different meanings is a source of confusion and a potential source of conflict between the constructive and classical mathematics communities. Since our goal here is a unification, and to bring classical mathematics into the world of constructive mathematics, I will be using new symbols for constructive logical operators. For constructively minded readers, please take careful note of the definitions of logical operators in Section 2.1.

Constructive mathematics also has new mathematical definitions that are analogous to common mathematical definitions but use these constructive logical operators, such as real numbers, continuity, and so on. Since this paper is about constructive mathematics, it would be tedious to add 'constructive' in front of every mathematical term. Instead, I will use unqualified terms, such as 'real numbers', for the constructive definitions, and use the prefix 'classical' when referring to 'classical real numbers'. I hope this will not be too confusing.

Note that due to space limitations, the appendices *Classical Connectives for Coq* and *Classical Values for Coq* are not included in this version of the paper, but they can be consulted in the preliminary version, which is available online at `http://arxiv.org/abs/1008.1213`.

2. Logic

My intention is to use dependent type theory viewed through the Curry–Howard isomorphism (Thompson 1991) as my deduction system for constructive logic. However, to

https://doi.org/10.1017/S0960129511000132 Published online by Cambridge University Press

help those unfamiliar with the constructive world and to illustrate how constructive logic is really an extension of classical logic, we will slowly and informally unveil dependent type theory, beginning with the classical fragment of constructive logic.

2.1. *Classical logic*

Before defining the logic, we will define a typed term language. To get us started, we will assume we are given types for booleans $\mathbb{B}$, natural numbers $\mathbb{N}$ and higher-order functions, $A \Rightarrow B$, over these basic types.

The booleans come with two constructors: true, $\mathsf{T} : \mathbb{B}$, and false, $\mathsf{F} : \mathbb{B}$. There is an if-then-else function for elimination:

$$\mathrm{if}_A : \mathbb{B} \Rightarrow A \Rightarrow A \Rightarrow A$$

If-then-else comes with its standard reduction rules:

$$\mathrm{if}\,\mathsf{T}xy \rightsquigarrow x$$
$$\mathrm{if}\,\mathsf{F}xy \rightsquigarrow y.$$

Using if-then-else, one can define any boolean operator.

The natural numbers come with constructors zero, $0 : \mathbb{N}$, and successor, $\mathsf{S} : \mathbb{N} \Rightarrow \mathbb{N}$, and an eliminator for primitive recursion:

$$\mathrm{rec}_A : \mathbb{N} \Rightarrow A \Rightarrow (\mathbb{N} \Rightarrow A \Rightarrow A) \Rightarrow A.$$

Primitive recursion comes with its standard reduction rules:

$$\mathrm{rec}_A\, 0zf \rightsquigarrow z$$
$$\mathrm{rec}_A(\mathsf{S}n)zf \rightsquigarrow fn(\mathrm{rec}_A\, nzf).$$

Higher-order functions will be defined by lambda expressions and come with the standard reduction rule:

$$(\lambda x.e)y \rightsquigarrow e[x \mapsto y].$$

With primitive recursion, we can define standard operations over natural numbers such as $+$ and $\times$. We can also define equality between natural numbers[†]:

$$0 =_{\mathbb{N}} 0 := \mathsf{T}$$
$$\mathsf{S}n =_{\mathbb{N}} 0 := \mathsf{F}$$
$$0 =_{\mathbb{N}} \mathsf{S}m := \mathsf{F}$$
$$\mathsf{S}n =_{\mathbb{N}} \mathsf{S}m := n =_{\mathbb{N}} m.$$

Equality for booleans can also be easily defined since it is just a boolean operator.

[†] Formally, equality is defined by

$$\lambda n.\, \mathrm{rec}_{\mathbb{N}\Rightarrow\mathbb{B}}\; n(\lambda m.\, \mathrm{rec}_{\mathbb{B}}\; m\mathsf{T}(\lambda m_0 c.\mathsf{F}))(\lambda n_0 r m.(\mathrm{rec}_{\mathbb{B}}\; m\mathsf{F}(\lambda m_0 c.rm_0)).$$

https://doi.org/10.1017/S0960129511000132 Published online by Cambridge University Press

We say two terms are **convertible** if they reduce to the same term. We write $t \leftrightarrow s$ to mean that the two terms t and s are convertible. Convertible terms denote the same value, so if $t \leftrightarrow s$, we can always replace t with s in any context.

For the logic, we will assume we are given symbols $\top$, $\bot$, $\wedge$, $\Rightarrow$[‡] and $\forall$. We will take as inference rules for these symbols the standard rules given by natural deduction (Troelstra and Schwichtenberg 1996). When a formula is derivable using the rules of our logic, we will say that the formula is **valid**, or the formula **holds**. This collection of classical connectives is well known to be complete. We can define the remaining logical connectives in terms of them:

$$\begin{aligned} \neg\varphi &:= \varphi \Rightarrow \bot \\ \varphi \vee \psi &:= \neg(\neg\varphi \wedge \neg\psi) \\ \exists a : A.\varphi(a) &:= \neg\forall a : A.\neg\varphi(a). \end{aligned}$$

We will call $\vee$ the **classical disjunction** or **weak disjunction**, and we will call $\exists$ the **classic existential** quantifier or **weak existential** quantifier.

For atomic relations, we define only one predicate symbol over the booleans, $\langle\cdot\rangle$:

$$\begin{aligned} \langle \mathsf{F} \rangle &:= \bot \\ \langle \mathsf{T} \rangle &:= \top. \end{aligned}$$

Notice that we are distinguishing between the type $\mathbb{B}$ of two elements and logical propositions. The $\langle\cdot\rangle$ relation acts as a kind of conversion from booleans to propositions.

We will not take any primitive notion of equality. Instead, we will use $\langle r =_{\mathbb{N}} s \rangle$ for the equality relation between two natural numbers and $\langle r =_{\mathbb{B}} s \rangle$ for the equality relation between two booleans.

We will assume we are given rules for case analysis and induction:

$$\frac{\theta(\mathsf{T}) \quad \theta(\mathsf{F})}{\forall b : \mathbb{B}.\theta(b)}$$

(a) Rule for case analysis.

$$\frac{\theta(0) \qquad \begin{array}{c}[n : \mathbb{N}; \theta(n)] \\ \vdots \\ \theta(\mathsf{S}n)\end{array}}{\forall n : \mathbb{N}.\theta(n)}$$

(b) Rule for induction.

To see that we have classical logic, we prove the law of excluded middle.

Theorem 1. (Law of excluded middle) For any proposition φ, $\varphi \vee \neg\varphi$ holds.

Proof. By the definition of $\vee$, we need to prove $\bot$ from the assumptions of $\neg\varphi$ and $\neg\neg\varphi$. But $\neg\neg\varphi = \neg\varphi \Rightarrow \bot$. By applying modus ponens to $\neg\varphi$ and $\neg\varphi \Rightarrow \bot$, we get $\bot$ as required. □

There is an objection that can be raised here. Even though we have proved the law of excluded middle, we still do not have a theorem giving us proof by contradiction. More

[‡] The context can be used to disambiguate whether $\Rightarrow$ refers to implication or a function type. We use the same symbol because in Section 3 we will actually identify implication with the function type.

https://doi.org/10.1017/S0960129511000132 Published online by Cambridge University Press

specifically, there is no proof schema for $\neg\neg\varphi \Rightarrow \varphi$ in our logic. This is not a problem because we can prove this for every formula in our classical fragment, specifically those formulas built from, $\top$, $\bot$, $\langle\cdot\rangle$, $\wedge$, $\Rightarrow$ and $\forall$, and hence the other connectives that are defined in terms of them.

Theorem 2 (Proof by contradiction). For any propositions φ and ψ and for any term r, the following hold:

(1) $\neg\neg\top \Rightarrow \top$.
(2) $\neg\neg\bot \Rightarrow \bot$.
(3) $(\neg\neg\varphi \Rightarrow \varphi) \Rightarrow (\neg\neg\psi \Rightarrow \psi) \Rightarrow (\neg\neg(\varphi \wedge \psi) \Rightarrow (\varphi \wedge \psi))$.
(4) $(\neg\neg\psi \Rightarrow \psi) \Rightarrow (\neg\neg(\varphi \Rightarrow \psi) \Rightarrow (\varphi \Rightarrow \psi))$.
(5) $\neg\neg\langle r\rangle \Rightarrow \langle r\rangle$.
(6) $(\forall a : A.\neg\neg\varphi(a) \Rightarrow \varphi(a)) \Rightarrow (\neg\neg(\forall a : A.\varphi(a)) \Rightarrow \forall a : A.\varphi(a))$.

Proof. (1)–(4) are easy constructive tautologies. By case analysis, (5) reduces to cases (1) and (2). For (6), we need to prove $\varphi(a)$ for an arbitrary a given our hypothesis. From $\forall a.\neg\neg\varphi(a) \Rightarrow \varphi(a)$ it suffices to prove $\neg\neg\varphi(a)$, but this follows easily from $\neg\neg(\forall a.\varphi(a))$. □

We say a proposition φ is **double-negation stable**, or simply **stable**, if $\neg\neg\varphi \Rightarrow \varphi$ holds. If φ is a stable formula, we are allowed to use proof by contradiction to prove it. By induction on the syntax of formulas, we see that every formula we can build so far is stable, and hence proof by contradiction is available to prove it. Notice that for $\varphi \Rightarrow \psi$ to be stable we only require that ψ be stable. Therefore, the hypotheses of a proposition are irrelevant when determining whether it is stable or not; only the conclusion matters.

With this fragment of our logic, we have now covered all of higher-order Peano arithmetic. This means that this fragment has more than enough logical power to develop essentially all of classical number theory using classical reasoning and even most of analysis (Simpson 1999). This way of defining classical logic inside constructive logic is known as the Gödel–Gentzen double-negation interpretation (Gödel 1933).

Now we turn our attention to extending this logic with constructive connectives.

2.2. *Constructive logic*

We will add two new primitive symbols: the constructive disjunction $+^{\dagger}$, and the constructive existential quantifier Σ. These connectives come with their standard rules of inference provided by natural deduction.

With the introduction of these two connectives, our inductive proof that $\neg\neg\varphi \Rightarrow \varphi$ now breaks down. We cannot expect $\neg\neg(\varphi + \psi) \Rightarrow (\varphi + \psi)$ to hold in general even when φ and ψ are stable. Neither can we prove in general that $\neg\neg(\Sigma n : \mathbb{N}.\varphi(n)) \Rightarrow (\Sigma n : \mathbb{N}.\varphi(n))$ holds even when $\varphi(n)$ is stable for all $n : \mathbb{N}$.

We say a proposition φ is **decidable** if $\varphi + \neg\varphi$ holds. Every decidable proposition is stable.

[†] The context will be used to disambiguate the constructive disjunction from addition.

https://doi.org/10.1017/S0960129511000132 Published online by Cambridge University Press

Recall equations (1) and (2), which state that our new constructive connectives can be weakened to their classical counterparts. This reflects part of the old view of constructive logic where every constructive proof is a classical proof. However, this old view is not fully internalised in constructive logic. We do not have that every proposition with constructive connectives implies the same proposition with the constructive connectives replaced by their classical counterparts.

Theorem 3. The formula

$$\neg(\forall n : \mathbb{N}.\Sigma m : \mathbb{N}.Rnm) \Rightarrow \neg(\forall n : \mathbb{N}.\exists m : \mathbb{N}.Rnm)$$

does not hold in general.

Proof. Given our deduction rules, we know that every valid formula holds in every topological model (Rasiowa. and Sikorski 1968). Therefore, to prove that this formula is not valid, it suffices to find a topological model where the formula does not hold.

In the topological model we will consider, propositions will represent open subsets of $\mathscr{C}$, the Cantor space. Let $f : \mathbb{N} \Rightarrow \mathscr{C}$ be an enumeration of some countable dense subset of the Cantor space. Let Rnm be clopen sets such that $\bigcup_{m\in\mathbb{N}} Rnm$ covers the entire Cantor space except $f(n)$. Thus $\Sigma m.Rnm$ represents $\mathscr{C}\backslash\{f(n)\}$, but $\exists m.Rnm$ represents $\mathscr{C}$. Therefore $\forall n.\Sigma m.Rnm$ represents $\varnothing$ while $\forall n.\exists m.Rnm$ represents $\mathscr{C}$. Thus $\neg(\forall n.\Sigma m.Rnm)$ represents $\mathscr{C}$ and $\neg(\forall n.\exists m.Rnm)$ represents $\varnothing$. Hence the formula is false in this model. □

Even adding an assumption that R is decidable does not help.

Theorem 4. The formula

$$(\forall nm : \mathbb{N}.Rnm + \neg Rnm) \Rightarrow \neg(\forall n : \mathbb{N}.\Sigma m : \mathbb{N}.Rnm) \Rightarrow \neg(\forall n : \mathbb{N}.\exists m : \mathbb{N}.Rnm)$$

does not hold constructively.

Proof. The topological model from Theorem 3 satisfies the hypothesis $\forall nm : \mathbb{N}.Rnm + \neg Rnm$, so the proof is the same as the proof of Theorem 3. □

2.3. *Reasoning with weak connectives*

We want to reason with the weak connectives as easily as we can reason with the connectives in traditional classical logic. Unfolding the definition of the weak disjunction to transform it into a negated conjunction is not a pleasant way of proceeding – we want to do case analysis. However, we cannot allow general case analysis to be done on the weak disjunction because that would imply that it is equivalent to the constructive disjunction.

Fortunately, there is a common situation when case analysis is allowed on the weak disjunction. When the goal is a stable proposition, case analysis can be done – see Figure 1.

It is easy to prove this rule is a tautology in constructive logic. A similar rule can be derived for the weak existential – see Figure 2.

In Coq, these two rules, when written as theorems, have a form close enough to that of an elimination rule of an inductive type that they can be used with Coq's `destruct t`

https://doi.org/10.1017/S0960129511000132 Published online by Cambridge University Press

$$\dfrac{\varphi \vee \psi \qquad \begin{matrix}[\varphi]\\ \vdots\\ \theta\end{matrix} \qquad \begin{matrix}[\psi]\\ \vdots\\ \theta\end{matrix} \qquad \begin{matrix}[\neg\neg\theta]\\ \vdots\\ \theta\end{matrix}}{\theta}$$

Fig. 1: A derived elimination rule for weak disjunction and stable θ.

$$\dfrac{\exists a : A.\varphi(a) \qquad \begin{matrix}[a : A; \varphi(a)]\\ \vdots\\ \theta\end{matrix} \qquad \begin{matrix}[\neg\neg\theta]\\ \vdots\\ \theta\end{matrix}}{\theta}$$

Fig. 2: A derived elimination rule for the weak existential and stable θ.

`using` tactic. This allows case analysis to be done on weak disjunctions and existentials almost as easily as they are done on constructive disjunctions and existentials. To make this easier still, we provide tactics[†] (`orWelim` and `existWelim`) that call `destruct t using` and then tries to solve the stability goal for θ automatically by searching a database of stability hints.

3. Dependent type theory

Up to this point, we have been taking a traditional view of logic where terms and formulas are separate entities. However, to proceed further, we will need to take a unified view of these two entities, and that unified view is provided by dependent type theory.

In dependent type theory, both data types and logical propositions are put on the same level – both are considered types. We define $\star$ as a kind of types[‡], and we can write $\mathbb{N} : \star$, or $\forall n : \mathbb{N}.\Sigma m : \mathbb{N}.Rnm : \star$. The values of propositional types are **proof objects**, or witnesses, of the proposition. For example, $\lambda x : \varphi.x$ has type $\varphi \Rightarrow \varphi$, and it is a proof object of this tautology. For every deduction of a formula θ, there is an associated proof object of type θ, which is given by the Curry–Howard isomorphism.

Dependent type theory enhances the elimination rules for logical connectives, allowing dependent elimination. For example, the dependent elimination rule for the constructive existential is shown in Figure 3.

Here, the proof object p is allowed to occur in the type of the conclusion, $\theta(p)$. Notice that when applying the elimination rule to a goal $\theta(p)$, the p occurring in θ is refined into the dependent pair $(a, \varphi(a))_{\Sigma\varphi}$. This is analogous to the induction rule for natural numbers where the n occurring in $\theta(n)$ is refined into 0 and $\mathsf{S}m$ in the two branches. In fact, it is completely analogous, so much so that we can define all the logical connectives

[†] See Appendix A in the preliminary version of this paper at `http://arxiv.org/abs/1008.1213`.

[‡] There are many variations of dependent type theory, which treat the types of types in various ways, and there are choices about whether quantification is predicative or impredicative. In this paper, we shall ignore these issues and leave it up to the reader to interpret this work as much as possible in whatever system they wish. For concreteness, this work has been prototyped in Coq – see Appendix A in the preliminary version of this paper at `http://arxiv.org/abs/1008.1213`.

https://doi.org/10.1017/S0960129511000132 Published online by Cambridge University Press

$$\frac{\Sigma a : A.\varphi(a) \quad \begin{array}{c}[a : A; \varphi(a)] \\ \vdots \\ \theta((a, \varphi(a))_{\Sigma\varphi}\end{array}}{\theta(p)}$$

Fig. 3: Dependent elimination rule for the constructive existential.

to be actual data types:

$$\begin{aligned} \top &:= () && \text{(unit type)} \\ \bot &:= \varnothing && \text{(void type)} \\ \varphi \wedge \psi &:= \varphi \times \psi && \text{(product type)} \\ \varphi \Rightarrow \psi &:= \varphi \Rightarrow \psi && \text{(function type)} \\ \forall x : A.\varphi(a) &:= \Pi x : A.\varphi(a) && \text{(dependent function type)} \end{aligned}$$

The classical formulas correspond to 'degenerate' data types, in the sense that if φ is a proposition made entirely from classical connectives[†], then for any two values of such a type, $p, q : \varphi$, we can prove $p \asymp q$ for a suitable notion of equivalence (see Section 3.1). Generally, we will use the logical notation for these classical formulas, and reserve the type theoretic notation for the other cases. However, technically, they will denote the same thing.

What we have called the constructive disjunction and existential are really the same things as the sum and dependent pair types, respectively, and we will continue to use the same notation for them both.

Since we have a kind for types, we can give signatures to predicates and relations. A predicate over A will have kind $A \Rightarrow \star$, and a binary relation on A and B will have kind $A \Rightarrow B \Rightarrow \star$, which is the same thing as a function from A to predicates on B. We already have an example of a predicate: $\lambda b : \mathbb{B}.\langle b \rangle : \mathbb{B} \Rightarrow \star$. For $b : \mathbb{B}$, we can now define $\langle b \rangle$ as

$$\langle b \rangle := \text{if } b \top \bot.$$

This basic predicate can be used to build other, and more complicated, predicates.

In exactly the same way, we can define families of types. For example, we can recursively define a family of types having $n : \mathbb{N}$ elements as a function of n:

$$\begin{aligned} \mathbb{Z}_0 &:= \varnothing \\ \mathbb{Z}_{\mathsf{S}n} &:= () + \mathbb{Z}_n. \end{aligned}$$

3.1. *Setoids*

So far we have not dealt with equality beyond the recursive definition of equality for inductive data types such as $\mathbb{N}$. In dependent type theory, it is common to use a structure

[†] In fact, all we need is for the conclusion to be composed of classical connectives.

https://doi.org/10.1017/S0960129511000132 Published online by Cambridge University Press

called a **setoid** that pairs a data type, called the carrier, with an equivalence relation over that type. The type of Setoids is denoted by Ω.

$$\Omega := \Sigma A : \star.\Sigma R : A \Rightarrow A \Rightarrow \star.\, \mathrm{EquivalenceRelation}\, R$$

When an $X : \Omega$ is used in a context expecting a type, we will leave the first projection implicit. We will write $x \asymp_X y$ when x and y are equivalent under the equivalence relation from the setoid X. We will omit the subscript when it can be inferred from the context.

All inductive data types form a natural setoid structure, where equality is the one defined in the usual recursive way. These natural setoid equivalences for the void data type $\varnothing$ and the unit data type () are trivial in the sense that equivalences hold for every pair of values in their respective types.

A function $f : X \Rightarrow Y$ between two setoids X and Y is respectful when it respects the equivalence relation from the two setoids:

$$\mathrm{Respectful}\, f := \forall xy : X.x \asymp y \Rightarrow fx \asymp fy.$$

The type of all respectful functions forms the (constructive) function space between setoids X and Y. We write $X \to Y$ for this type:

$$X \to Y := \Sigma f : X \Rightarrow Y.\, \mathrm{Respectful}\, f.$$

In fact, we will make $X \to Y$ a setoid. We define two respectful functions $f, g : X \to Y$ to be equivalent if they are equivalent pointwise:

$$f \asymp g := \forall x : X.fx \asymp gx.$$

Propositions also form a setoid. Two propositions $\varphi, \psi : \star$ are considered equivalent if they are logically equivalent:

$$\varphi \asymp \psi := (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi).$$

Taking this together with respectful functions, we can form respectful predicates of type $X \to \star$ and higher arity respectful relations.

Given a setoid X and a respectful predicate on that setoid, $P : X \to \star$, we can define a subsetoid with carrier $\Sigma a : X.Pa$, where equality is defined by equality of X on the first projection:

$$(x, p) \asymp_{\Sigma P} (y, q) := x \asymp_X y.$$

Given a setoid X and a respectful equivalence relation on that setoid E, a quotient setoid can be formed by replacing the equivalence relation on X with E:

$$x \asymp_{X/E} y := Exy.$$

In this paper, we will also be interested in **stable setoid**s, which are setoids that have an additional property that the equivalence relation is stable[†]. Setoids for inductive data types are stable setoids, and respectful functions whose codomain are stable setoids are also stable setoids. One can define a setoid of (constructive) real numbers (O'Connor 2008a), $\mathbb{R}$, and this setoid is stable.

[†] Stable setoids could reasonably be called Hausdorff spaces (Bauer and Taylor 2009).

https://doi.org/10.1017/S0960129511000132 Published online by Cambridge University Press

3.2. *Notation*

We will use some compressed notation for a few common idioms. In particular, we will identify sets with respectful predicates, so we will use set notation to define and work with predicates:

$$\begin{aligned}
\wp X &:= X \to \star \\
x \in X &:= Xx \\
\{x : X | \varphi(x)\} &:= \lambda x : X.\varphi(x) \\
\{fx | x : X\} &:= \{y | \Sigma x : X.y \asymp fx\} \\
X \cup Y &:= \{x | x \in X + x \in Y\} \\
X \cap Y &:= \{x | x \in X \wedge x \in Y\} \\
X \subseteq Y &:= \forall x.x \in X \Rightarrow x \in Y.
\end{aligned}$$

We also define some shorthand for quantifiers:

$$\begin{aligned}
\forall x \in X.\varphi(x) &:= \forall x.x \in X \Rightarrow \varphi(x) \\
\Sigma x \in X.\varphi(x) &:= \Sigma x.x \in X \wedge \varphi(x) \\
\forall n < m.\varphi(n) &:= \forall n.\langle n < m\rangle \Rightarrow \varphi(n) \\
\Sigma n < m.\varphi(n) &:= \Sigma n.\langle n < m\rangle \wedge \varphi(n).
\end{aligned}$$

and, similarly, for $\exists$ and other inequalities.

Finally, we define unique existence for setoids:

$$\exists! x : X.\varphi(x) := (\exists x : X.\varphi(x)) \wedge (\forall xy : X.\varphi(x) \Rightarrow \varphi(y) \Rightarrow x \asymp y).$$

In most cases we will leave the first projection of the dependent pair type (which are also known as the constructive existential) implicit in the same way as I have already mentioned for the case of using setoids in a context requiring a type.

I will also not explicitly write out proof objects that go into the construction of constructive existentials. Instead, I will just give the witness and will leave it to the reader to fill in the proof.

4. Infinite pigeonhole principle

Let us consider a real life example of using classical reasoning in the development of constructive mathematics. In this example, we will be using the infinite pigeonhole principle. The infinite pigeonhole principle says that if you distribute an infinite number of pigeons amongst a finite number of pigeonholes, then there is some pigeonhole with an infinite number of pigeons. There is no constructive proof of the infinite pigeonhole principle because finding a pigeonhole with an infinite number of pigeons is undecidable in general. Therefore, we state the theorem using classical existentials:

$$\forall n : \mathbb{N}.\forall f : \mathbb{N} \Rightarrow \mathbb{Z}_n.\exists m.\forall a : \mathbb{N}.\exists b > a.\langle fb =_{\mathbb{Z}_n} m\rangle.$$

Because this theorem is stated in the classical fragment of our logic, we can use classical reasoning to prove this theorem.

https://doi.org/10.1017/S0960129511000132 Published online by Cambridge University Press

We will now look at a constructive context where this classical theorem can be used.

4.1. *Hausdorff metrics*

I have used this pigeonhole principle during my development of constructively compact sets to prove that the Hausdorff metric is, in fact, a metric (O'Connor 2008b; 2009). A metric on a type X is defined by a ball relationship $B^X : \mathbb{Q}^+ \Rightarrow X \Rightarrow X \Rightarrow \star$, where $B^X_\varepsilon ab$ means that $dab \leqslant \varepsilon$. The ball relationship is used instead of the distance function for a couple of reasons. First, it allows us to define a metric without having to construct the real numbers. Second, it allows us to define metrics for spaces without a computable distance function.

Given a metric space X, one can define a metric on $\mathfrak{F}X$, the finite sets of X, using the Hausdorff metric. Given two finite sets $\mathscr{A}$ and $\mathscr{B}$, we defined the ball relation as

$$B^{\mathfrak{F}X}_\varepsilon \mathscr{A}\mathscr{B} := \big(\forall a \in \mathscr{A}.\exists b \in \mathscr{B}.B^X_\varepsilon ab\big) \wedge \big(\forall b \in \mathscr{B}.\exists a \in \mathscr{A}.B^X_\varepsilon ab\big).$$

This says that for every point in $\mathscr{A}$ there is some point in $\mathscr{B}$ that lies within ε and *vice versa*. Notice the use of the weak existential here – it may not be immediately clear why we need to use the weak existential rather than the constructive one, so to help understand why this is the case, we consider the case where X is the real numbers $\mathbb{R}$.

If we were using the traditional metric space definition, the distance between two sets of real numbers $\mathscr{A}$ and $\mathscr{B}$ would be

$$d\mathscr{A}\mathscr{B} := \max\left\{\max_{a\in\mathscr{A}}\min_{b\in\mathscr{B}} dab, \max_{b\in\mathscr{B}}\min_{a\in\mathscr{A}} dab\right\}.$$

Since $\mathscr{A}$ and $\mathscr{B}$ are finite sets, and min and max are constructive functions, this definition is constructive. However, notice that $\min_{b\in\mathscr{B}} dab$ does not compute the point b at which the minimum value is obtained. If such a b were computable, we could decide for any two real numbers if $a \leqslant b$ or $b \leqslant a$. However, this is well known to be undecidable.

Since we want to represent this metric using our ball relation, it would be inappropriate to use constructive existence in our Hausdorff metric definition since that would require us to know at which point b the minimum $\min_{b\in\mathscr{B}} dab$ is obtained, and this would make the definition too strong.

4.2. *Applying the pigeonhole principle*

One key property required of the ball relation is the closedness property:

$$\forall \varepsilon : \mathbb{Q}^+.\big(\forall \delta : \mathbb{Q}^+.B^X_{\varepsilon+\delta} ab\big) \Rightarrow B^X_\varepsilon ab.$$

We need to prove this property for our definition of the Hausdorff metric (given that the metric space X already has this property). Consider for the moment one half of the definition of the Hausdorff metric:

$$\forall a \in \mathscr{A}.\exists b \in \mathscr{B}.B^X_\varepsilon ab.$$

https://doi.org/10.1017/S0960129511000132 Published online by Cambridge University Press

To prove half of the closeness property, it suffices to prove

$$\forall \varepsilon : \mathbb{Q}^+. \left(\forall n : \mathbb{N}^+. \forall a \in \mathscr{A}. \exists b \in \mathscr{B}. B^X_{\varepsilon+\frac{1}{n}} ab\right) \Rightarrow \forall a \in \mathscr{A}. \exists b \in \mathscr{B}. B^X_\varepsilon ab.$$

For any given $a \in \mathscr{A}$, we need to prove there is some $b \in \mathscr{B}$ such that $B^X_\varepsilon ab$ holds. The trouble is that the b found by our hypothesis depends on n, and we need to find one b that holds for every n.

This is where the infinite pigeonhole principle comes into play. There is going to be some b such that that

$$B^X_{\varepsilon+\frac{1}{n}} ab$$

holds for infinitely many n since $\mathscr{B}$ is a finite set. Because of the weakening property of metric spaces, this one b works for all n. Thus $B^X_\varepsilon ab$ holds as needed. The other half of the definition of the Hausdorff metric is proved similarly. Notice that because we have used the weak existential in the definition of the Hausdorff metric, we are able to apply the infinite pigeonhole principle, which is a result in classical logic.

One might object to this whole formulation of the problem because, though it is technically done in constructive logic, we have avoided any real constructive work by avoiding the constructive existential. In other words, there is no computational content in any of this work.

While not denying this argument, it is important to see that we still maintain the constructive content of the rest of the theory. While there is no computational content in this fragment of the development, once the Hausdorff metric is defined, we can go on to define compact sets as the Cauchy completion of the finite sets.

The completion of a metric space X (O'Connor 2008a) is represented by coherent functions from $\mathbb{Q}^+$ to X. A function representing a point y from the completion of X takes a tolerance $\varepsilon : \mathbb{Q}^+$ and returns a value in X that is within ε of the point y. A function is **coherent** if it represents some point. Two different functions that both represent the same point are equivalent under a suitable setoid equivalence relation. We see that the completion does have constructive content, even if the underlying metric is in the classical fragment of constructive logic. For example, for each compact subset of the plane, we can compute a finite set of points that approximates the compact set.

Figure 4 shows the output of a computation inside Coq (not extracted from Coq) of an approximation of the graph of the function $\lambda x. \sin(3x)$ on the interval $[-1, 1]$. Here the finite set is being represented as a boolean matrix together with coordinates for the upper-left and lower-right corners of the matrix. Using Coq's notation mechanism, we display the boolean matrix using single Unicode characters denoting whether the value of the corresponding entry of the matrix is `true` or `false`. This produces the convenient visualisation of the finite set seen in the figure and illustrates how concrete computation can really proceed even when the metric is in the classical fragment of our logic.

5. Double negation

There are alternative definitions of the classical disjunction and existential in terms of their constructive counter parts. By slapping a double negation on the outside of a constructive

https://doi.org/10.1017/S0960129511000132 Published online by Cambridge University Press

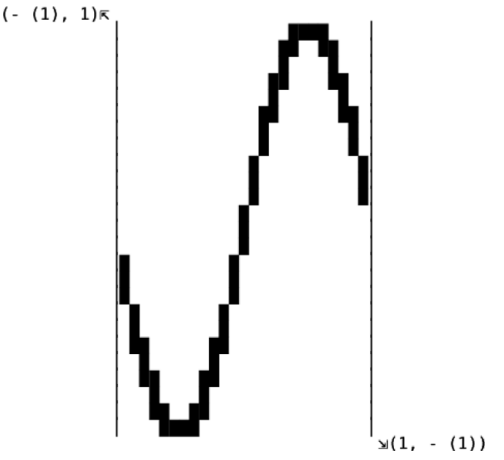


Fig. 4: The result of computing an approximation of the graph of $\sin(3x)$ on $[-1, 1]$.

connective, it can be turned into a classical connective.

$$\begin{aligned} \varphi \vee \psi &:= \neg\neg(\varphi + \psi) \\ \exists x.\varphi(x) &:= \neg\neg(\Sigma x.\varphi(x)). \end{aligned}$$

These definitions are equivalent to the definitions given previously in Section 2.1, but the observation that you use double negations to transform constructive statements into classical ones is very powerful owing to the special properties of double negation. In particular, double negation forms a monad, and thus monadic style reasoning can be used.

5.1. *Monads*

Monads are a standard construction in category theory (Kock 1972). In this section, we review the basics of monads from a functional programming perspective (Wadler 1995). A (strong) monad $\mathfrak{M} : \Omega \Rightarrow \Omega$ is a setoid constructor that comes with three polymorphic operations:

$$\begin{aligned} \mathrm{unit}_{\mathfrak{M}} &: \Pi X.X \to \mathfrak{M}X \\ \mathrm{map}_{\mathfrak{M}} &: \Pi X Y.(X \to Y) \to \mathfrak{M}X \to \mathfrak{M}Y \\ \mathrm{join}_{\mathfrak{M}} &: \Pi X.\mathfrak{M}(\mathfrak{M}X) \to \mathfrak{M}X. \end{aligned}$$

These operations need to satisfy certain laws. There are several different ways of phrasing the monad laws, but one of the easiest is to first define two auxiliary combinators. The first combinator, $\mathrm{bind}_{\mathfrak{M}}$ takes a function $f : X \to \mathfrak{M}Y$ and lifts the argument, returning a function of type $\mathfrak{M}X \to \mathfrak{M}Y$:

$$\mathrm{bind}_{\mathfrak{M}}\, f := \mathrm{join}_{\mathfrak{M}} \circ (\mathrm{map}_{\mathfrak{M}}\, f).$$

https://doi.org/10.1017/S0960129511000132 Published online by Cambridge University Press

The second combinator takes $f : X \to \mathfrak{M}Y$ and $g : Y \to \mathfrak{M}Z$ and combines them to get a function of type $X \to \mathfrak{M}Z$:

$$g \lll f := (\mathrm{bind}_{\mathfrak{M}}\, g) \circ f.$$

Using this combinator, the monad laws can be stated as

$$\begin{aligned} f \lll \mathrm{unit}_{\mathfrak{M}} &\asymp f \\ \mathrm{unit}_{\mathfrak{M}} \lll f &\asymp f \\ f \lll (g \lll h) &\asymp (f \lll g) \lll h. \end{aligned}$$

Let us call a function whose type is of the form $X \to \mathfrak{M}Y$ a Kleisli function. As you can see, these Kleisli functions compose together nicely. By composing a regular function of type $X \to Y$ together with $\mathrm{unit}_{\mathfrak{M}}$, it can be transformed into Kleisli function.

We will use the following notation for these operations when it is clear which monad is under consideration:

$$\begin{aligned} \bar{f} &:= \mathrm{map}_{\mathfrak{M}}(f) \\ \check{f} &:= \mathrm{bind}_{\mathfrak{M}}(f) \\ \hat{a} &:= \mathrm{unit}_{\mathfrak{M}}(a). \end{aligned}$$

There are several other useful combinators for monads. One called ap, takes a function 'inside a monad', $f : \mathfrak{M}(X \to Y)$, and a value $x : \mathfrak{M}X$ 'applies' f to x and returns a value of type $\mathfrak{M}Y$. We write ap using the infix operator @:

$$f @ x := \mathrm{bind}_{\mathfrak{M}}(\lambda f_0.\overline{f_0}x)f.$$

Ap can be used to create higher arity map functions, such as

$$\mathrm{map}\,2_{\mathfrak{M}} : (X \to Y \to Z) \to \mathfrak{M}X \to \mathfrak{M}Y \to \mathfrak{M}Z$$

$$\mathrm{map}\,2_{\mathfrak{M}} f x y := \hat{f} @ x @ y.$$

We will give $\mathrm{map}\,2_{\mathfrak{M}}$ a special notation when applied to infix operators:

$$x \bar{\diamond} y := \mathrm{map}\,2_{\mathfrak{M}}(\lambda ab.a \diamond b)xy.$$

With these combinators, together with other combinators and notations, manipulating monadic values can be almost as pleasant as manipulating regular values.

5.2. *The double-negation monad*

Double negation $\mathfrak{N} : \star \Rightarrow \star$ is a type constructor:

$$\mathfrak{N}\varphi := \neg\neg\varphi.$$

This is trivially a setoid constructor because every function to $\varnothing$[†] is respectful and equivalent to every other function of the same type.

[†] Recall that $\neg\varphi := \varphi \Rightarrow \bot$ and that $\bot := \varnothing$, so a value of type $\neg\varphi$ is really a function to the void type.

https://doi.org/10.1017/S0960129511000132 Published online by Cambridge University Press

$$\dfrac{\mathfrak{N}\varphi \qquad \begin{array}{c}[\varphi]\\ \vdots\\ \theta\end{array} \qquad \begin{array}{c}[\neg\neg\theta]\\ \vdots\\ \theta\end{array}}{\theta}$$

Fig. 5: Elimination rule for the double-negation monad and stable θ.

The operations for this monad correspond to to the natural proofs of the appropriate type:

$$\begin{aligned}\mathrm{unit}_{\mathfrak{N}} &:= \lambda\varphi : \star.\lambda a : \varphi.\lambda k : \neg\varphi.ka\\ \mathrm{map}_{\mathfrak{N}} &:= \lambda\varphi\psi : \star.\lambda f : \varphi \Rightarrow \psi.\lambda x : \mathfrak{N}\varphi.\lambda k : \neg\psi.x(k \circ f)\\ \mathrm{join}_{\mathfrak{N}} &:= \lambda\varphi : \star.\lambda x : \mathfrak{N}(\mathfrak{N}\varphi).\lambda k : \neg\varphi.x(\mathrm{unit}_{\mathfrak{N}}\, k).\end{aligned}$$

Any double-negated type, $\mathfrak{N}\varphi$, is automatically stable independent of whether φ uses constructive connectives or not. Indeed, $\mathrm{join}_{\mathfrak{N}}$ witnesses the proof of stability. This means that weak disjunction and weak existentials can be eliminated nicely whenever one is trying to prove a goal of type $\mathfrak{N}\theta$. This is actually a special instance of a more general rule that says that if the goal is stable, a hypothesis of the form $\mathfrak{N}\varphi$ can be transformed into a hypothesis of the form φ.

In the special case where θ is of the form $\mathfrak{N}\theta_0$, the proof of this elimination rule is witnessed by the $\mathrm{bind}_{\mathfrak{N}}$ function. Indeed, if θ is stable, then θ and $\mathfrak{N}\theta$ are isomorphic, with $\mathrm{unit}_{\mathfrak{N}}$ witnessing $\theta \Rightarrow \mathfrak{N}\theta$, and the proof of stability witnessing $\mathfrak{N}\theta \Rightarrow \theta$. Using this isomorphism, we can prove our elimination rule for the double-negation monad by transforming θ into $\mathfrak{N}\theta$, using $\mathrm{bind}_{\mathfrak{N}}$, and transforming $\mathfrak{N}\theta$ back to θ.

The advantage of this monadic approach is that not only can classical connectives for disjunction and existence be defined, but arbitrarily complex constructive propositions can be turned into classical-like propositions.

5.3. *Hybrid systems*

Geuvers *et al.* (2010) developed a framework for proving the safety of hybrid discrete and continuous systems. For example, they prove the correctness of a simple thermostat hybrid system by proving that it maintains a minimum temperature.

This work uses exact real arithmetic through the constructive real numbers, rather than floating point numbers, in order to ensure correctness. The difficulty here is that many propositions about real numbers are undecidable and hence not provable in constructive mathematics. For example, constructive trichotomy $\forall xy : \mathbb{R}.(x < y) + (x \asymp y) + (y < x)$, and even constructive dichotomy $\forall xy : \mathbb{R}.(x \leqslant y) + (y \leqslant x)$ do not hold, but they do hold when the constructive disjunction is replaced by the weak disjunction or when the lemma conclusions are put into the double-negation monad.

Geuvers *et al.* use the double-negation monad instead of classical connectives. The use of the double-negation monad allows one to use trichotomy and dichotomy when reasoning about the hybrid systems. Using the classical fragment of constructive logic is acceptable because the ultimate goal of a safety proof is to show that the set of reachable

https://doi.org/10.1017/S0960129511000132 Published online by Cambridge University Press

states is contained within some compact set, and being a member of a compact set is a stable relation.

5.4. *Feit–Thompson Theorem*

The Feit–Thompson Theorem, also known as the Odd-order Theorem, states that any group of odd order is solvable. The revised paper proof is about 255 pages long, and there is currently an ambitious effort underway to formalise the proof in Coq (Gonthier *et al.* 2007).

The ultimate goal of the Feit–Thompson Theorem is double-negation stable, so classical reasoning can be allowed. Since large parts of the proof only involve reasoning over finite sets, restricting oneself to constructive reasoning is not actually restrictive because most classical laws hold constructively in the finite case. However, a few parts of the proof involve reasoning about undecidable predicates. For these cases, a modal operator called `classically` has been defined[†]:

$$\texttt{classically}\varphi := \forall b : \mathbb{B}.(\varphi \Rightarrow \langle b \rangle) \Rightarrow \langle b \rangle.$$

Notice that this definition is logically equivalent to the double-negation monad defined in Section 5.2. The reason this form of the double-negation monad is used is that, for this particular project, one very often has goals of the form $\langle r \rangle$. When this is the case, a hypothesis of $\texttt{classically}\varphi$ can be immediately applied and a new hypothesis φ introduced into the context. Using SSReflect tactics (Gonthier and Stéphane 2009), this can be done easily in one step. This leaves the user to prove their original goal $\langle r \rangle$ under the hypothesis φ with the `classically` stripped away.

6. Classical functions

In classical mathematics, functions are represented by a certain class of binary predicates over the domain and codomain. The constraints are that for each point in the domain there is at at least one and at most one related point in the codomain:

$$(\forall x : X.\exists Y.Fxy) \wedge (\forall yz : B.Fxy \Rightarrow Fxz \Rightarrow y \asymp z).$$

If we drop the constraint that there is at least one related point in the codomain, we get the definition of a partial function.

Defining this notion of a classical function using constructive mathematics is not difficult. The problem is that we want to be able to compose and manipulate functions easily, and using relations makes this difficult. For example, to write $F(Gx) \asymp G(Fx)$ using binary predicates, one would have to give names to the intermediate values and write out something like

$$\exists y_1 y_2 z_1 z_2.Gxy_1 \wedge Fy_1z_1 \wedge Fxy_2 \wedge Gy_2z_2 \wedge z_1 \asymp z_2.$$

[†] Technically, the definition of `classically` is a little different to the one we give here, but the spirit is close enough for our purposes.

https://doi.org/10.1017/S0960129511000132 Published online by Cambridge University Press

Translating from functional notation to binary predicate notation is quite painful and greatly expands the formula. In first-order logic, there is not much else that can be done without extending the language with new function symbols. However, with dependent type theory there is an easy transformation.

Using curried functions, one would write the type of a binary predicate on X and Y as $X \to Y \to \star$. Binary predicates can also be equivalently viewed as having type $X \to \wp Y$, which is a function from X to unary predicates on Y.

If the binary predicate F satisfies the properties of being a function and it is applied to an input $x : X$, the resulting unary predicate will have some special properties. The predicate $Fx : \wp Y$ is a predicate that holds for at least one value and at most one value. For a classical partial function F, the predicate Fx will hold for at most one value.

We define a **classical partial value**, or a **weak partial value**, of a stable setoid Y as a subsetoid of predicates on Y that holds for at most one value and is also stable. We denote the stable setoid of weak partial values over Y as $\mathfrak{P}Y$:

$$\mathfrak{P}Y := \Sigma \mathscr{Y} : \wp Y.(\forall y.\neg\neg y \in \mathscr{Y} \Rightarrow y \in \mathscr{Y}) \wedge (\forall y_1 y_2.y_1 \in \mathscr{Y} \Rightarrow y_2 \in \mathscr{Y} \Rightarrow y_1 \asymp y_2).$$

We define a **classical value**, also called a **weak value**, of stable setoid Y as a subsetoid of $\mathfrak{P}Y$ that holds for at least one value. We denote this stable setoid of weak values over Y as $\mathfrak{V}Y$:

$$\mathfrak{V}Y := \Sigma \mathscr{Y} : \mathfrak{P}Y.\exists y.y \in \mathscr{Y}.$$

We use the weak existential here because we want to define functions for classical mathematics. If we had used the constructive existential, our definition would have ended up almost equivalent to a constructive function.

Now a weak (partial) function can simply be defined as a function that produces weak (partial) values. Once we have shown that weak (partial) values form a monad (see Section 6.1), we will be able to see that a weak (partial) function is simply a Kleisli function for the weak (partial) value monad.

Has this shuffling around of definitions really helped us? The answer is yes, because both $\mathfrak{P}$ and $\mathfrak{V}$ are monads (see Section 6.1), users can use standard monadic combinators and programing techniques to manipulate functions and values of this type. Consider the example from before of trying to state $F(Gx) \asymp G(Fx)$. Using the the weak value monad, the functions F and G will have type $X \to \mathfrak{V}X$. Using the bind operation, we can write the statement as $\check{F}(Gx) \asymp \check{G}(Fx)$. This statement is almost as easy to write as the classical notation, and certainly much easier than the long-winded definition using binary predicates.

6.1. *Monadic operations*

Recalling our notation from Section 3.2, we define $\mathrm{unit}_{\mathfrak{P}} : X \to \mathfrak{P}X$ by

$$\mathrm{unit}_{\mathfrak{P}}\, x_0 := \{x : X | x \asymp x_0\}.$$

Given a function $f : X \to Y$, we define $\mathrm{map}_{\mathfrak{P}}\, f : \mathfrak{P}X \to \mathfrak{P}Y$ by

$$\mathrm{map}_{\mathfrak{P}}\, f \mathscr{X} := \{y : Y | \exists x \in \mathscr{X}.fx \asymp y\}.$$

https://doi.org/10.1017/S0960129511000132 Published online by Cambridge University Press

Finally, we define $\mathrm{join}_{\mathfrak{P}} : \mathfrak{P}(\mathfrak{P}X) \to \mathfrak{P}X$ by

$$\mathrm{join}_{\mathfrak{P}}\, \mathscr{X} := \mathscr{X} \circ \mathrm{unit}_{\mathfrak{P}}\,.$$

The definitions of $\mathrm{map}_{\mathfrak{V}}$, $\mathrm{join}_{\mathfrak{V}}$ and $\mathrm{unit}_{\mathfrak{V}}$ are similar.

Recall from Section 2.1 that we defined a function from booleans to propositions, $\lambda b.\langle b\rangle : \mathbb{B} \to \star$. There is a similar function taking classical boolean values to propositions. For $b : \mathfrak{V}\mathbb{B}$, we use the same notation to transform it into a proposition, and the context will make it clear which of these is meant:

$$\langle b\rangle := \mathsf{T} \in b.$$

6.2. *Choice functions*

The constructive axiom of choice is stated using the constructive quantifiers as follows:

$$\forall AB : \star.(\forall x : A.\Sigma y : B.Rxy) \Rightarrow (\Sigma f : A \Rightarrow B.\forall x : A.Rx(fx)).$$

This is a theorem of dependent type theory (for example, it is provable in MLTT (Nordström *et al.* 1990), CIC (Coq Development Team 2009) and OTT (Altenkirch and McBride 2006)), and I will call this the 'constructive theorem of choice'. Notice, however, that the function f will not, in general, respect the equivalence relations between A and B if A and B are setoids.

Using the classical connectives and classical functions, we can state the classical axiom of choice for setoids X and Y and a respectful relation R:

$$(\forall xy.\neg\neg Rxy \Rightarrow Rxy) \Rightarrow (\forall x : X.\exists y : Y.Rxy) \Rightarrow (\exists f : X \to \mathfrak{V}Y.\forall x : X.fx \subseteq Rx).$$

This classical choice axiom is double-negation stable by construction, and it is not a theorem of dependent type theory (for example, it is not a theorem of MLTT, CIC or OTT). This is one significant difference from the usual approach of doing classical mathematics in a constructive logic by adding additional classical axioms. In the usual approach, the axiom of choice is inherited from the constructive theorem of choice.

However, it is worth noting that the classical axiom of choice is a consistent negative statement. In OTT, one can add consistent axioms to propositions without destroying canonicity of the non-propositional fragment (Altenkirch *et al.* 2007). Thus, in OTT, users have the flexibility to choose whether or not to include the axiom of choice in their work.

6.2.1. *Definite choice.*

Although the general axiom of choice is not available without the use of other axioms, the principle of definite choice holds:

$$(\forall xy.\neg\neg Rxy \Rightarrow Rxy) \Rightarrow (\forall x : X.\exists! y : Y.Rxy) \Rightarrow (\Sigma f : A \to \mathfrak{V}(B).\forall x : A.fx \subseteq Rx).$$

Notice that the existence of the classical function is even constructive.

The proof is pretty trivial because the hypothesis of definite choice contains exactly the ingredients needed to build the weak function f. Given an arbitrary x, we have that Rx is a stable predicate, and classically there exists a unique y such that $y \in Rx$ holds. Thus we can create a function $f : A \Rightarrow \mathfrak{V}B$ such that $\forall x.A.fx \subseteq Rx$ holds. Because R is respectful, we can prove this function is respectful too.

https://doi.org/10.1017/S0960129511000132 Published online by Cambridge University Press

6.3. *Examples of classical functions*

One example of a classical function is the characteristic of a ring structure. For every ring R there is a monoid homomorphism from $\rho : \mathbb{N} \to R$ defined by $\rho n := n \cdot 1_R$. The characteristic of R is the GCD of the set $\{n | \rho n \asymp 0_R\}$.

In general, the characteristic of a ring is uncomputable, even if the ring enjoys decidable equality. However, we can create a classical function for it. More generally, we can create a classical function to compute the GCD of any subset of $\mathbb{N}$. First, we give the specification for the GCD:

$$\begin{aligned} \mathrm{gcdSpec} &: \wp\mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \star \\ \mathrm{gcdSpec}\, P x &:= (\forall y \in P.\langle x|y\rangle) \wedge (\forall z.(\forall y \in P.\langle z|y\rangle) \Rightarrow \langle z|x\rangle). \end{aligned}$$

Notice that $\mathrm{gcdSpec}\, P x$ is a stable proposition because $x|y$ is a decidable relation. We can also find a proof object for the proposition that there is a unique number n satisfying $\mathrm{gcdSpec}\, P$:

$$\mathrm{gcdExists} \ : \ \forall P \ : \wp\mathbb{N}.\exists! n : \mathbb{N}.\, \mathrm{gcdSpec}\, P n.$$

Because we have used the classical existential here, the entire theorem lies in the classical fragment of our logic. Therefore, we can complete the proof using any of the usual classical techniques. Using definite choice, we can define the $\mathrm{gcd} : \wp\mathbb{N} \to \mathfrak{V}\mathbb{N}$ operation.

We can prove various lemmas about this classical function. For example,

$$\forall P Q.\langle \mathrm{gcd}(P \cup Q)\overline{|}\, \mathrm{gcd}\, P\rangle.$$

It is now easy to define the characteristic of a ring:

$$\mathrm{char}\, R := \mathrm{gcd}\{n | \rho(n) \asymp_R 0\}.$$

Another example of a classical function is the degree function for polynomials. When the ring of coefficients enjoys decidable equality, the degree function is a fine constructive function. However, when decidable equality does not hold for the ring structure, as is the case for the real numbers, the degree function can no longer be constructed. However, even in this case, it is still important to be able to talk about bounds on degrees of polynomials in theorems.

In *A Course in Constructive Algebra* (Mines *et al.* 1988) the authors write (pp. 60–61):

> For $n \in \mathbb{N}$, a polynomial f in $R[X]$ that can be written as $\sum_{i=0}^{n-1} r_i X^i$ is said to have **degree at most** $n-1$, written $\deg f \leqslant n-1$ or $\deg f < n$. [...]. If $r_i \neq 0$ for some $i \geqslant d$, then we say that f has **degree at least** d, and write $\deg f \geqslant d$. If $\deg f \leqslant d$ and $\deg f \geqslant d$, then we say that f has **degree** d, written $\deg f = d$, [...]
>
> If f and g are polynomials, then we write $\deg f \leqslant \deg g$ if $\deg g < n$ implies $\deg f < n$ for each $n \in \mathbb{N}$; and we write $\deg f < \deg g$ if $\deg g \leqslant n+1$ implies $\deg f < n$ for each $n \in \mathbb{N}$. [...]

Notice all the effort that needs to go into defining all the possible ways of using and comparing degrees of polynomials. There are so many, it is unreasonable for the authors to write them all out. Indeed, it only takes them five lines before they write:

> [...] Then ab is the leading coefficient of fg, and $\deg fg = \deg f + \deg g$.

https://doi.org/10.1017/S0960129511000132 Published online by Cambridge University Press

However, they have not defined what $\deg h = \deg f + \deg g$ means, although the reader can make up a reasonable definition on the spot.

Instead, classical functions can be used to give a rigorous definition of degree. We can define a specification for degree in a similar way to how we specified the GCD:

$$\mathrm{degSpec}\, f\, d := \left(\sum_{i=0}^{d} r_i X^i \asymp f\right) \wedge \left(\forall n. \sum_{i=0}^{n} r_i X^i \asymp f \Rightarrow \langle d \leqslant n \rangle\right).$$

This specification is stable if the underlying ring R is a stable setoid. In particular, this is the case for $\mathbb{R}$.

We can prove that for every polynomial f, classically there exists a unique d such that $\mathrm{degSpec}\, f\, d$ holds. From this proof, we can create a weak function $\deg : R[X] \to \mathfrak{V}\mathbb{N}$. Using our monodic combinators, we can construct arbitrary propositions about degrees such as

$$\begin{aligned} &\langle \deg f \ \bar{\leqslant}\ \hat{n} \rangle \\ &\langle \deg f \ \bar{\leqslant}\ \deg g \rangle \\ &\langle \deg f \ \bar{=}\ \deg g \mathbin{\bar{+}} \deg h \rangle. \end{aligned}$$

This technique can express most of the propositions about degrees required, but there is a caveat. *A Course in Constructive Algebra* defines $\deg f \geqslant n$ using constructive existentials. Therefore, there is no way to replicate this statement in this framework. However, I believe that the use of such constructive statements about degrees is relatively rare. In those cases where a constructive statement is required, one would need to revert to the old method of making an *ad hoc* definition.

7. Conclusion

One advantage of founding an interactive theorem prover on dependent type theory is that it has the capability of supporting both constructive and classical reasoning. However, we do not need to add axioms that destroy canonicity to get classical reasoning. Classical reasoning was already there because it is a fragment of constructive reasoning. This allows one to retain the benefits of constructive logic while still being able to derive classical results. We have shown three examples of how to integrate traditionally classical results with constructive mathematics.

We also created a monadic method of defining classical functions spaces, again without the need to add classical axioms, and presented two examples of classical functions that can be used in constructive mathematics.

Maintaining canonicity in dependent type theory is important because it allows for computation to be done inside the logic using a technique known as reflection (Barendregt and Geuvers 2001). It also allows for program extraction, which means software can be developed along with proofs of their correctness. Dependent pair types (that is, constructive existentials) are important in the development of correct software because they allow us to tie invariants right into data types (Leroy 2009).

https://doi.org/10.1017/S0960129511000132 Published online by Cambridge University Press

Prototype Coq modules for the weak connectives and weak value monads are given in Appendix A and Appendix B, respectively, of the preliminary version of this paper[†]. The GCD example has been formalised in Coq using these modules and the lemma that $\forall P\,Q.\langle \gcd(P \cup Q)\overline{|}\,\gcd\,P\rangle$ has been verified in order to show this proof of concept. Although the monadic functions work well for creating statements using classical functions, further theory of weak values needs to be developed and included in the module in order to make reasoning about them easier.

I hope that the idea of embedding classical reasoning in this way will be developed further for proof assistants. While we have seen that it is possible to do classical reasoning, more support in constructive proof assistants would be needed if classically minded users are to be able to use them transparently for classical reasoning. With enough support, it should be possible for users to not even know that the underlying logic of the system is constructive. This would benefit constructive users as well, because they would be able to use classical results in their work.

References

Altenkirch, T. and McBride, C. (2006) Towards observational type theory. Manuscript, available online.

Altenkirch, T., McBride, C. and Swierstra, W. (2007) Observational equality, now! In: *PLPV '07 Proceedings of the 2007 workshop on Programming languages meets program verification*, ACM 57–68.

Barendregt, H. and Geuvers, H. (2001) Proof-assistants using dependent type systems. In: Robinson, A. and Voronkov, A. (eds.) *Handbook of Automated Reasoning* **2**, Elsevier Science Publishers 1149–1238.

Bauer, A. and Taylor, P. (2009) The Dedekind reals in abstract Stone duality. *Mathematical Structures in Computer Science* **19** 757–838.

Coq Development Team (2009) *The Coq Proof Assistant Reference Manual – Version V8.2*, INRIA-Rocquencourt. (Available at `http://coq.inria.fr`.)

Geuvers, H., Koprowski, A., Synek, D. and van der Weegen, E. (2010) Automated machine-checked hybrid system safety proofs. In: Kaufmann, M. and Paulson, L. (eds.) Proceedings Interactive Theorem Proving, ITP 2010. *Springer-Verlag Lecture Notes in Computer Science* **6172** 259–274.

Gonthier, G., Mahboubi, A., Rideau, L., Tassi, E. and Théry, L. (2007) A modular formalisation of finite group theory. In: Schneider, K. and Brandt, J. (eds.) Proceedings Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007. *Springer-Verlag Lecture Notes in Computer Science* **4732** 86–101.

Gonthier, G. and Stéphane, L. R. (2009) An Ssreflect Tutorial. Technical Report RT-0367, INRIA. (Available at `http://hal.inria.fr/inria-00407778/en/`.)

Gödel, K. (1933) Zur intuitionistischen Arithmetik und Zahlentheorie. *Ergebnisse eines mathematischen Kolloquiums* **4** 34–38. (English translation: On intuitionistic arithmetic and number theory. In: M. Davis (ed.) *The Undecidable – Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*, Raven, New York, 1965 (Reprint, Dover, 2004) 75–81.)

[†] The preliminary version of this paper is available at `http://arxiv.org/abs/1008.1213`.

https://doi.org/10.1017/S0960129511000132 Published online by Cambridge University Press

Kock, A. (1972) Strong functors and monoidal monads. *Archiv der Mathematik* **23** 113–120.

Leroy, X. (2009) Programming with dependent types: Passing fad or useful tool? (Available at `http://www.comlab.ox.ac.uk/ralf.hinze/WG2.8//26/slides/xavier.pdf`.)

Mines, R., Richman, F. and Ruitenburg, W. (1988) *A Course in Constructive Algebra*, Springer-Verlag.

Nordström, B., Petersson, K. and Smith, J. M. (1990) *Programming in Martin-Löf's type theory: an introduction*, Clarendon Press.

Norell, U. (2007) *Towards a practical programming language based on dependent type theory*, Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden.

O'Connor, R. (2008a) Certified exact transcendental real number computation in Coq. In: Mohamed, O. A., Muñoz, C. and Tahar, S. (eds.) Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2008). *Springer-Verlag Lecture Notes in Computer Science* **5170** 246–261.

O'Connor, R. (2008b) A computer verified theory of compact sets. In: Buchberger, B., Ida, T. and Kutsia, T. (eds.) Proceedings of Austrian–Japanese Workshop on Symbolic Computation in Software Science (SCSS 2008). RISC-Linz Report Series (08-08) 148–162.

O'Connor, R. (2009) *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*, Ph.D. thesis, Radboud Universiteit Nijmegen.

Rasiowa., H. and Sikorski, R. (1968) *The Mathematics of Metamathematics* (second edition), Polish Scientific Publishers.

Simpson, S. G. (1999) *Subsystems of second order arithmetic*, Perspectives in Mathematical Logic, Springer-Verlag.

Thompson, S. (1991) *Type Theory and Functional Programming*, Addison Wesley.

Troelstra, A. S. and Schwichtenberg, H. (1996) *Basic Proof Theory*, Cambridge tracts in theoretical computer science **43**, Cambridge University Press.

Wadler, P. (1995) Monads for functional programming. In: Jeuring, J. and Meijer, E. (eds.) Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques – Tutorial Text. *Springer-Verlag Lecture Notes in Computer Science* **925** 24–52.

https://doi.org/10.1017/S0960129511000132 Published online by Cambridge University Press