BOURBAKI'S FORMAL SYSTEM IN HASKELL

ALEX NELSON

Abstract. We implement the abstract syntax tree and rudimentary syntactic support for the formal language found in Bourbaki's *Theory of Sets* [1]. Although we do not implement any of the deductive apparatus, it should be simple enough for a motivated reader. **Caution:** If you are trying to run this on a computer with less than 16 TB of RAM, then you should expect to wait a long time for it to finish.

Contents

Date: January 7, 2024.

1. Formal Language of Bourbaki

Bourbaki's formal system is rather difficult to understand, since it's jettisoned almost immediately after construction, and uses many idiosyncratic terms. My reference will be the English translation published by Springer, the softcover reprint.[1] Aitkens's commentary [2] is also worth consulting. The basic "Rosetta stone" of terminology appears to be:

Bourbaki	$\approx$	Modern Terminology
Sign	$\approx$	Letter (of a fixed ambient alphabet)
Assembly	$\approx$	String (over the ambient alphabet)
Letter	$\approx$	Variable
Specific Sign	$\approx$	Primitive notion (of a theory)
Relation	$\approx$	Logical formula
Formative Criteria	$\approx$	Formal grammar for well-formed formulas

Some terms have no modern translation, like "logical sign" appears to refer to "primitive notions in their underlying logic".

We will hide *and* from Prelude, since it is more natural to introduce a function which is Bourbaki's conjunction operator.

```
import Data.Set hiding (cartesianProduct)
import Prelude hiding (and)
```

Bourbaki's "letter" is what we would call a "variable". I'm going to encode it as an arbitrary string.

```
type Letter = String
```

Bourbaki's "term" resembles what we think of terms (namely, they're "mathematical objects" as opposed to propositions). However, Bourbaki uses Hilbert's ε-calculus, which has fallen into relative obscurity. Complicating matters, Bourbaki uses a convoluted system of "linkages" to avoid distinguishing *bound variables* from *free variables*.

The basic idea of Hilbert's ε-calculus can be understood piecemeal. First, we think of a predicate in first-order logic as being a term of type

```
type Predicate = Term → Formula  {-intuition, not actual code -}
```

Then we can understand a "choice operator" as taking a predicate; if there is an object which satisfies that predicate, then the choice operator returns it. If there is no object which satisfies the predicate, then an arbitrary-but-fixed object is returned. Hilbert uses $\varepsilon_x P[x]$ as the notation for this term. Bourbaki sometimes uses $\tau_x P[x]$ and other times replaces all instances of x by a box $\square$, then draws "linkages" (i.e., lines) from those boxes to the τ. This is rather difficult to typeset. Instead, we will use de Bruijn levels[2], and call the bound de Bruijn level a *TBox* keeping track of the depth and the variable it replaced.

Bourbaki also introduces the notation for substituting a term T for a variable x in an expression S by $(T \mid x)S$. We will add this to the abstract syntax tree encoding for a term. Later, we will create a typeclass for syntactic classes in Bourbaki's system which support substitutions, in order to *actual perform a substiution.*

[1] Apparently this is the English translation dated 1968 of the French 1970 edition. How this time-traveling is possible, well, that's beyond my understanding.

[2] The difference between a de Bruijn level and index depends on where you start counting.

```
data Term = TTau Integer Letter Relation
  | TBox Integer Letter
  | TVar Letter
  | TSubst Term Letter Term
  | TPair Term Term
  deriving (Show, Eq)
```

The notion of a "formula" in Bourbaki is called a "relation", which is perhaps an unfortunate choice of words.

Bourbaki works with an adequate set of connectives, namely disjunction $A \vee B$ and negation $\neg A$. The other connectives are just abbreviations for expression; in (I §1.1) example 1, Bourbaki quickly mentions in as obscure a manner as possible that:

$$A \implies B \quad := \quad (\neg A) \vee B. \tag{1a}$$

In (I §3.4), Bourbaki defines conjunction as:

$$A \wedge B \quad := \quad \neg((\neg A) \vee (\neg B)). \tag{1b}$$

In (I §3.5), Bourbaki defines "equivalence" (bi-conditional) as:

$$A \iff B \quad := \quad (A \implies B) \wedge (B \implies A). \tag{1c}$$

We introduce helper functions to improve the readability of encodings.

We can substitute a term for a variable in a relation, which Bourbaki denotes by $(T \mid x)A$ where T is a term and A is a relation. Like we did for terms, we are forming an abstract syntax tree for relations, and we have a node encoding this.

The only primitives in Bourbaki's system of set theory are equality of terms $t_1 = t_2$ and set membership $t_1 \in t_2$.

```
data Relation = ROr Relation Relation
  | RNot Relation
  | RSubst Term Letter Relation
  | REq Term Term
  | RIn Term Term
  deriving (Show, Eq)

and         :: Relation → Relation → Relation
and      a b = RNot (ROr (RNot a) (RNot b))

implies     :: Relation → Relation → Relation
implies a b = ROr (RNot a) b

iff         :: Relation → Relation → Relation
iff      a b = and (implies a b) (implies b a)
```

1.1. **Substitutions.** Now we can introduce a type class which abstracts the notion of *performing substitutions*. This is justified by formative criteria CF8 from (I §1.4) which states that the assembly $(T \mid x)A$ is a term when A is a term, and it's a relation when A is a relation.

```
class Subst a where
  subst :: Letter → Term → a → a
```

When we work with terms, we can consider the following cases:

(1) $(T \mid x)y = \begin{cases} T & \text{if } x = y \\ y & \text{otherwise} \end{cases}$
(2) $(T \mid x)\tau_x A = \tau_x A$ since x no longer appears in $\tau_x A$
(3) $(T \mid x)\tau_y A = \tau_y((T \mid x)A)$ if $y \neq x$ (and we use the notion of substitution in a relation)
(4) $(T \mid x)\square = \square$ since $\square$ is "just" a constant term expression

As far as $(T \mid x)\big((T' \mid y)T''\big)$ for terms T', T'' and variable y, this requires a bit of care. If $x = y$, then nothing is done. On the other hand, if $x \neq y$, criteria CS2 (I §1.2) tells us how to "commute" substitutions:

$$(B \mid x)(C \mid y)A = ((B \mid x)C \mid y)(B \mid x)A. \tag{2}$$

This gives us enough information to define substitution for terms:

```
instance Subst Term where
  subst y t t' = case t' of
    (TBox _ _) → t'
    (TVar x) → if x ≡ y
                 then t
                 else t'
    (TSubst b x a) → if x ≡ y
                       then (TSubst (subst y t b) x a)
                       else (TSubst (subst y t b) x (subst y t a))
    (TTau n x p) → if x ≡ y
                     then t'
                     else (TSubst t y t')
    (TPair t1 t2) → TPair (subst y t t1) (subst y t t2)
```

When we work with relations, criteria of substitution CS5 from (I §1.2) gives us the explicit definition for almost all relations:

(1) $(T \mid x)(A \vee B) = ((T \mid x)A) \vee ((T \mid x)B)$
(2) $(T \mid x)(\neg A) = \neg((T \mid x)A)$
(3) $(T \mid x)(t_1 = t_2) \quad = \quad ((T \mid x)t_1) = ((T \mid x)t_2)$
(4) $(T \mid x)(t_1 \in t_2) = ((T \mid x)t_1) \in ((T \mid x)t_2)$

Bourbaki also includes in CS5 instructions for the derived connectives $(T \mid x)(A \implies B)$, $(T \mid x)(A \wedge B)$, $(T \mid x)(A \iff B)$, but these are not needed.

```
instance Subst Relation where
  subst y t (ROr a b) = ROr (subst y t a) (subst y t b)
  subst y t (RNot a)  = RNot (subst y t a)
  subst y t (RSubst b x r) = if y ≡ x then (RSubst b x r)
                                      else RSubst (subst y t b) x (subst y t r)
  subst y t (REq a b) = REq (subst y t a) (subst y t b)
  subst y t (RIn a b) = RIn (subst y t a) (subst y t b)
```

1.2. **Simplification.** As far as actually *simplifying* expressions, we have this operation abstracted away in its own typeclass.

```
class Simplifier a where
  simp :: a → a
```

There are a few sources of simplification of formulas: performing substitutions, replacing $A \vee \neg A$ with a simpler tautology, and eliminating double negatives.

```
instance Simplifier Relation where
  simp (ROr a b) = let a' = simp a
                       b' = simp b
                   in if (simp (RNot a')) ≡ b'
                      then (REq (TVar "_") (TVar "_"))
                      else if a' ≡ b'
                      then a'
                      else ROr a' b'
  simp (RNot (RNot a)) = simp a
  simp (RNot a)  = RNot (simp a)
  simp (RSubst t x r)     = simp $ subst x t r
  simp (REq a b) = let a' = simp a
                       b' = simp b
                   in if a' ≡ b'
                      then REq (TVar "_") (TVar "_")
                      else REq (simp a) (simp b)
  simp (RIn a b)  = RIn (simp a) (simp b)
```

Simplifying terms boils down to performing substitutions. Variables and bound variables ($TBox$) are in simplest form.

```
instance Simplifier Term where
  simp (TTau m x r) = TTau m x (simp r)
  simp (TBox m x)   = TBox m x
  simp (TVar x)     = TVar x
  simp (TSubst t x b) = simp $ subst x t b
  simp (TPair a b)    = TPair (simp a) (simp b)
```

1.3. ***Deductive System.** Just a few remarks about the "deductive system" Bourbaki uses. Specifically, Bourbaki uses a Hilbert proof calculus, but not for first-order logic. Instead Bourbaki uses Hilbert's ε-calculus. Consequently, there are only two inference rules given (I §2.2):

- (a_1) Any instance of an axiom may be used at any time in a proof;
- (a_2) Any instance of a "scheme" may be used at any time in a proof;
- (b) *Modus Ponens*: if in previous proof steps A and $A \implies B$ have been established, then we may write down B in a proof step.

Axioms (I §2.1) are either "explicit axioms" (which is what we normally think of when defining a new gadget) or "implicit axioms", which are obtained by applying a scheme. Schemes are "rules" which constructs a formula—Bourbaki is vague about its meaning. Derived inference rules are given in items labeled $C1$, $C2$, $C3$, ...

The axioms Bourbaki gives may be found summarized in the very last page of the book. The first four are the so-called "Russell–Bernays axioms"[3] (I §3.1) where $A \implies B$ is understood as an abbreviation for $(\neg A) \vee B$:

[3]This appears to be the axioms found in the *Principia Mathematica*, specifically corresponding to axioms ∗1.2, ∗1.3, ∗1.4, and ∗1.6 in *Principia*. Bernays proved its logical completeness in

(S1) $(A \vee A) \implies A$
(S2) $A \implies (A \vee B)$
(S3) $(A \vee B) \implies (B \vee A)$
(S4) $(A \implies B) \implies ((C \vee A) \implies (C \vee B))$.

Then axioms are given for quantified theories (I §4.2) as:

(S5) If R is a relation of theory $\mathscr{T}$, if T is a term in $\mathscr{T}$, and if x is a letter, then the relation $(T \mid x)R \implies (\exists x)R$ is an axiom.

The last two logical axioms concern equality (I §5.1):

(S6) Let x be a letter, let T and U be terms in theory $\mathscr{T}$, and let $R[x]$ be a relation in $\mathscr{T}$. Then the relation $(T = U) \implies (R[T] \iff R[U])$ is an axiom.
(S7) If R and S are relations in a theory $\mathscr{T}$, and if x is a letter, then the relation $((\forall x)(R \iff S)) \implies (\tau_x(R) = \tau_x(S))$ is an axiom.

The usual quantifier introduction and elimination rules are given as derived inference rules: S5 is $\exists$-introduction, C27 is $\forall$-introduction, and C30 is $\forall$-elimination. Existential-elimination can be given automatically using the τ-operator to obtain the witness term.

2. Epsilon Calculus Implementation

2.1. **De Bruijn levels.** We don't actually need to keep track of which object a $\tau_x A$ refers to. We encode the $\square$ using de Bruijn levels. As a consistency check, we keep track of the variable being bound as well as the depth of the τ (which will match the de Bruijn level).

```
class Shift a where
  shift :: a → a
```

For terms, this amounts to adding 1 to the level of τ and $\square$ instances. For substitutions, this requires shifting in both the body and the term being substituted in.

```
instance Shift Term where
  shift (TTau m x r) =  TTau (m + 1) x r
  shift (TBox m x)   =  TBox (m + 1) x
  shift (TVar x)     =  TVar x
  shift (TSubst b x a) = TSubst (shift b) x (shift a)
  shift (TPair a b)  =  TPair (shift a) (shift b)
```

For relations, this "descends" the syntax tree to terms, which are then shifted.

```
instance Shift Relation where
  shift (ROr a b) = ROr (shift a) (shift b)
  shift (RNot a) = RNot (shift a)
  shift (RSubst a x r) = RSubst (shift a) x (shift r)
```

"Axiomatische Untersuchungen des Aussagen-Kalkuls der *Principia Mathematica*." *Mathematische Zeitschrift* **25** (1926) 305–320; translated into English in Richard Zach's *Universal Logic: An Anthology* (2012) pp.43–58. Russell and Whitehead call these axioms "principle of tautology", "principle of addition", "principle of permutation", "principle of summation". Coincidentally, this is also the axioms found in Hilbert and Ackermann's *Grundzüge der theoretischen Logik* (1928).

```
shift (REq a b) = REq (shift a) (shift b)
shift (RIn a b) = RIn (shift a) (shift b)
```

2.2. **Tau operator.** The $\tau_x R$ can be formed using this helper function *tau x R*, which will handle the substitution of $\square$ for x in R (along with all necessary shifting).

```
tau      :: Letter → Relation → Term
tau x r = TTau 0 x $ subst x (TBox 0 x) (shift r)
```

2.3. **Logical quantifiers.** We can introduce logical quantifiers (with some simplification handled automatically) since Bourbaki follows Hilbert and defines

$$\exists x.A[x] \quad := \quad A[\tau_x A[x]] \tag{3}$$

and by de Morgan's law,[4]

$$\forall x.A[x] \quad := \quad A[\tau_x \neg A[x]]. \tag{4}$$

But since I'm more skeptical of accidentally writing some kind of bug, I'm just going to use $\neg(\exists x.\neg A[x])$ as the definition for the universal quantifier. This gives us the code:

```
exists      :: Letter → Relation → Relation
exists x r  = simp $ subst x (tau x r) r

for_all     :: Letter → Relation → Relation
for_all x r = simp $ RNot (exists x (RNot r))
```

Note: the ε-calculus is responsible for the ridiculously large sizes of the assemblies, specifically because we are using these definitions of quantifiers. One bit of low-hanging fruit would be to introduce one of these quantifiers as a primitive, and define the other in terms of the identity $\neg(\exists x.\neg P[x]) \iff \forall x.P[x]$ or $\neg(\forall x.\neg P[x]) \iff \exists x.P[x]$. We would also need to add rules to the simplifier to rewrite

$$P[\tau_x P[x]] \mapsto \exists x.P[x]$$

and

$$P[\tau_x \neg P[x]] \mapsto \forall x.P[x].$$

If we were to add axioms to support this, I suppose (since the first four axioms describing propositional logic appear to be from Hilbert and Ackermann, we can continue this path) we would follow Hilbert and Ackermann's *Grundzüge der theoretischen Logik* (1928):

(1) $(\forall x.P[x]) \implies P[x]$
(2) $P[x] \implies (\exists x.P[x])$.

We would add the inference rules:

(1) If x is not free in φ and we have proven $\varphi \implies \psi[x]$, then we can infer $\varphi \implies \forall x.\psi[x]$;
(2) If we have proven $\psi[x] \implies \varphi$, then we can infer $(\exists x.\psi[x]) \implies \varphi$.

[4] If we let $B[x] = \neg A[x]$, and using de Morgan's law $\neg(\exists x \neg A[x]) \iff \forall x.A[x]$, then $\neg(\exists x \neg A[x]) \iff \neg(\exists x.B[x]) \iff \neg B[\tau_x B[x]] \iff \neg\neg A[\tau_x B[x]]$. Double negation simplifies this to $\forall x.A[x] \iff A[\tau_x \neg A[x]]$.

3. Fresh Variables for Assemblies

3.1. **Set of all variables.** We need to form the set of all variables (including, for the sake of caution, the variables which were captured by τ expressions).

```
class Vars a where
  vars :: a → Set Letter
```

For terms, this operation just descends to $\square$ and letters, removing any variables which are substituted out. Since we use *tau* to perform the choice operation, substitutions should have already occurred.

```
instance Vars Term where
  vars (TTau _ x r)   = Data.Set.union (Data.Set.singleton x) (vars r)
  vars (TBox _ x)     = (Data.Set.singleton x)
  vars (TVar x)       = Data.Set.singleton x
  vars (TSubst b x a) = Data.Set.delete x (Data.Set.union (vars a) (vars b))
  vars (TPair a b)    = Data.Set.union (vars a) (vars b)
```

For relations, this just descends down to terms, and form the unions of the subtrees. As for terms, upon the substitution nodes we simply remove the variable being replaced by terms. (And, as for terms, this shouldn't really occur since simplification will perform the replacement.)

```
instance Vars Relation where
  vars (ROr a b)      = Data.Set.union (vars a) (vars b)
  vars (RNot a)       = vars a
  vars (RSubst a x r) = Data.Set.delete x (Data.Set.union (vars a) (vars r))
  vars (REq a b)      = Data.Set.union (vars a) (vars b)
  vars (RIn a b)      = Data.Set.union (vars a) (vars b)
```

3.2. **Fresh Variables.** Given a set of variables V, and some variable we'd like to use x, we will check if $x \in V$ and if so try some variant of x to see if it occurs in V. This is done by adding a subscript x_n where n is an integer we increment upon trying again.

```
freshVar :: Letter → Int → Set Letter → Letter
freshVar x m vs = if (x ++ (show m)) `Data.Set.member` vs
                  then freshVar x (m + 1) vs
                  else x ++ (show m)
```

Now, for any Haskell expression which is an instance of the *Vars* typeclass, we can find a fresh variable for it. This checks if the variable x appears in the set of variables; if not, then just use it. Otherwise, we need to find a "fresher" version of the variable (by appending a numeric value "subscript" to it).

```
fresh :: Vars a ⇒ Letter → a → Letter
fresh x fm = let vs = vars fm
             in if x ∈ vs
                then freshVar x 0 vs
                else x
```

4. Length of terms

4.1. **Counting the occurrences of a variable.** How many times does a variable occur in an expression? We can count this, using a typeclass.

```
class Occur a where
  occur :: Letter → a → Integer
```

Now, x doesn't appear in $\tau_x R$, so its occurrences should short-circuit to zero. But if somehow it gets through, we should count x appearing zero times in $\square$ bound variables.

For substitutions, there is some subtlety here, which is a source of bugs in naive implementations. Observe, if $x = y$, then $(B \mid x)A$ will replace all n instances of x in A by B. But if B has m instances of x, then we get $m \cdot n$ instances of x in the substitution $(B \mid x)A$.

However, when $x \neq y$, then $(B \mid y)A$ will replace all n_y instances of y in A by B. When there are m instances of x in B, this results in an additional $n_y m$ instances of x in $(B \mid y)A$. When there are n_x instances of x in A *before substitution*, then we have a total of $n_y m + n_x$ occurrences of x in $(B \mid y)A$.

```
instance Occur Term where
  occur x (TTau _ y r)   = if x ≡ y then 0 else (occur x r)
  occur x (TBox _ _)     = 0
  occur x (TVar y)       = if x ≡ y then 1 else 0
  occur x (TSubst b y a) = if x ≡ y
                           then (occur x b) * (occur x a)
                           else (occur x b) * (occur y a) + (occur x a)
  occur x (TPair a b)    = (occur x a) + (occur x b)
```

For relations, the same subtlety surrounding occurrences of a variable in a substitution (but the same reasoning holds for relations as for terms). In all other cases, it boils down to counting the occurrences in the subtrees, and adding them all together in the end.

```
instance Occur Relation where
  occur x (ROr a b)      = (occur x a) + (occur x b)
  occur x (RNot a)       = occur x a
  occur x (RSubst a y r) = if x ≡ y
                           then (occur x a) * (occur x r)
                           else (occur x a) * (occur y r) + (occur x r)
  occur x (REq a b)      = (occur x a) + (occur x b)
  occur x (RIn a b)      = (occur x a) + (occur x b)
```

4.2. **Length of assemblies.** Now we come to the main part of the original motivation for this program, what is the length of an assembly? For any assembly A, we will write $|A|$ for the length of the assembly A. We have a typeclass abstracting this notion:

```
class Len a where
  len :: a → Integer
```

For terms, we have the inductive definition:

(1) $|\tau_x R| = 1 + |R|$
(2) $|\square| = 1$
(3) $|x| = 1$
(4) $|(B \mid x)A| = (|B| - 1) \cdot o(x, A) + |A|$ where $o(x, A)$ is the number of occurrences of x in A; if one is suspicious of this claim, it's because $|(B \mid x)A| = |B| \cdot o(x, A) + (|A| - o(x, A))$, and then simple algebra gives us the result.
(5) $|\langle A, B\rangle| = 1 + |A| + |B|$ since we are using the "original" convention that $\supset t_1\ t_2$ is an ordered pair, which just prepends the concatenation of strings with one symbol.

instance *Len Term* **where**

```
len (TTau _ _ r)   = 1 + len r
len (TBox _ _)     = 1
len (TVar _)       = 1
len (TSubst b x a) = ((len b) − 1) ∗ (occur x a) + (len a)
len (TPair a b)    = 1 + (len a) + (len b)
```

For relations, we have

(1) $|A \vee B| = 1 + |A| + |B|$
(2) $|\neg A| = 1 + |A|$
(3) $|(B \mid x)R| = (|B| - 1)o(x, R) + |R|$ where $o(x, R)$ is the number of occurrences of the variable x in the relation R
(4) $|A = B| = 1 + |A| + |B|$
(5) $|A \in B| = 1 + |A| + |B|$

instance *Len Relation* **where**

```
len (ROr a b)      = 1 + len a + len b
len (RNot a)       = 1 + len a
len (RSubst a y r) = ((len a) − 1) ∗ (occur y r) + (len r)
len (REq a b)      = 1 + len a + len b
len (RIn a b)      = 1 + len a + len b
```

5. Set Theory

Caution: the code we implement assumes we are working with sentences, i.e., formulas with no free variables. This is fine for our purposes, but we should include code to make sure the variables we're quantifying over are fresh. This adds needless overhead to our implementation, and adds no benefit.

After C49 in (II §1.4), Bourbaki introduces the notation $\mathcal{E}_x(R)$ for

> To represent the term $\tau_y(\forall x)((x \in y) \iff R)$ which does not depend on the choice of y (distinct from x and not appearing in R), we shall introduce a functional symbol $\mathcal{E}_x(R)$; the corresponding term does not contain x. This term is denoted by 'the set of all x such that R'.

We denote this by *ens x R*.

```
ens :: Letter → Relation → Term
ens x r = let y = fresh "y" r
          in tau y (for_all x (iff (RIn (TVar x) (TVar y)) r))
```

The unordered pair is introduced in (II §1.5), defined as $\mathcal{E}_z(x = z \vee y = z)$ which is then abbreviated as $\{x, y\}$. This exists and is unique by the second axiom of Bourbaki's set theory, which means it really is a well-defined notion.

```
  -- The set with two elements, a.k.a., the unordered pair.
pair      :: Term → Term → Term
pair x y = let z = fresh "z" (REq x y)
           in ens z (ROr (REq x (TVar z)) (REq y (TVar z)))
```

5.1. **Ordered Pairs.** This is formalized in (II §2) of Bourbaki's *Theory of Sets*. Bourbaki defines $\supset T\ U$ for the ordered pair of T and U as a primitive notion. But we can use the usual set-theoretic construction of ordered pairs. Purists can modify code in the way following explicit instructions.

Now, before we can define the ordered pair using the familiar set-theoretic definition $(x, y) = \{\{x\}, \{x, y\}\}$, we need to define an unordered singleton.

```
ssingleton    :: Term → Term
ssingleton x = let z = fresh "z" x
               in ens z (REq x (TVar z))
```

Now, for ordered pairs, we use the set-theoretic definition for debugging purposes (if you wanted to use the original Bourbaki formulation, you can replace this line of code with the primitive *TPair* data constructor)

```
  -- use orderedPair = TPair for debugging purposes
orderedPair   :: Term → Term → Term
orderedPair   = TPair
  -- orderedPair x y = pair (ssingleton x) (pair x y)
orderedTriple :: Term → Term → Term → Term
orderedTriple x y z = orderedPair (orderedPair x y) z
```

5.2. **Cartesian Product of Sets.** The Cartesian product of sets is defined in (II §2.2) Definition 1 as

$$X \times Y := \mathcal{E}_z\big((\exists x)(\exists y)(z = (x, y) \wedge x \in X \wedge y \in Y)\big). \tag{5}$$

The implementation follows this definition:

```
cartesianProduct :: Term → Term → Term
cartesianProduct x y = ens "z" (exists "x'"
                                  (exists "y'"
                                    (and (REq (TVar "z")
                                              (orderedPair (TVar "x'") (TVar "y'")))
                                         (and (RIn (TVar "x'") x)
                                              (RIn (TVar "y'") y)))))
```

5.3. **Subsets.** In (II §1.2), Definition 1, Bourbaki defines the predicate for a subset $X \subset Y$ as:

$$X \subset Y \quad := \quad \forall z(z \in X \implies z \in Y). \tag{6}$$

We use this definition in the construction:

```
subset :: Term → Term → Relation
subset u v = for_all "s" (implies (RIn (TVar "s") u) (RIn (TVar "s") v))
```

5.4. **Empty set.** The empty set is defined as $\tau_X((\forall y)(y \notin X))$ in comments towards the end of (II §1.7), and we use this as the definition for it:

```
emptySet :: Term
emptySet = tau "X" (for_all "y" (RNot (RIn (TVar "y") (TVar "X"))))
```

5.5. **Cardinality of sets.** In (III §3.1), Bourbaki defines the notion of "the cardinal of a set" using equipotential sets. Two sets A and B are called equipotent, denoted by Bourbaki as $\mathrm{Eq}(A, B)$, if there exists a bijection between sets A and B. Then the cardinality of a set A is defined as $\mathrm{card}(A) := \tau_Z(\mathrm{Eq}(A, Z))$. But in a footnote, Bourbaki tells us the explicit definition for $1 := \mathrm{card}(\{\emptyset\})$. It's tedious:

$$\begin{aligned}\tau_Z\Big((\exists u)(\exists U)\Big(u = (U, \{\emptyset\}, Z) \text{ and } U \subset \{\emptyset\} \times Z \\ \text{and } (\forall x)\big((x \in \{\emptyset\}) \implies (\exists y)((x, y) \in U)\big) \\ \text{and } (\forall x)(\forall y)(\forall y')\big(((x, y) \in U \text{ and } (x, y') \in U) \implies (y = y')\big) \\ \text{and } (\forall y)((y \in Z) \implies (\exists x)((x, y) \in U))\Big)\Big)\end{aligned} \tag{7}$$

This allows us to find the primitive definition of $\mathrm{card}(A)$:

$$\begin{aligned}\mathrm{card}(A) := \tau_Z\Big((\exists u)(\exists U)\Big(u = (U, A, Z) \text{ and } U \subset A \times Z \\ \text{and } (\forall x)\big((x \in A) \implies (\exists y)((x, y) \in U)\big) \\ \text{and } (\forall x)(\forall y)(\forall y')\big(((x, y) \in U \text{ and } (x, y') \in U) \implies (y = y')\big) \\ \text{and } (\forall y)((y \in Z) \implies (\exists x)((x, y) \in U))\Big)\Big)\end{aligned} \tag{8}$$

Here is where all the low-hanging fruit for optimization occurs, we could use different definitions of cardinality. There are five terms in this definition contained in the scope of the outer two universal quantifiers $\forall u \forall U(\dots)$ which we define as *termA*, *termB*, *termC*, *termD*, and *termE*. We faithfully write the code for this convoluted definition:

```
termA :: Term → Letter → Letter → Letter → Relation
termA x u upperU z = REq (TVar u) (orderedTriple (TVar upperU) x (TVar z))

termB :: Term → Letter → Letter → Relation
termB x upperU z = subset (TVar upperU) (cartesianProduct x (TVar z))

termC :: Term → Letter → Relation
termC x upperU = for_all "x" (implies (RIn (TVar "x") x)
                                      (exists "y" (RIn (orderedPair (TVar "x") (TVar "y"))
```

```
                                                    (TVar upperU))))
termD :: Letter → Relation
termD upperU = for_all "x"
  (for_all "y" (for_all "z"
    (implies (and (RIn (orderedPair (TVar "x") (TVar "y")) (TVar upperU))
                  (RIn (orderedPair (TVar "x") (TVar "z")) (TVar upperU)))
             (REq (TVar "y") (TVar "z")))))
termE :: Letter → Letter → Relation
termE upperU z = for_all "y" (implies
  (RIn (TVar "y") (TVar z))
  (exists "x" (RIn (orderedPair (TVar "x") (TVar "y"))
                   (TVar upperU))))
card :: Term → Term
card x = tau "Z" (exists "u" (exists "U" (and (termA x "u" "U" "Z")
                                              (and (termB x "U" "Z")
                                                   (and (termC x "U")
                                                        (and (termD "U")
                                                             (termE "U" "Z"))))))))
```

As examples of this definition, Bourbaki defines 1 and 2 as

```
one :: Term
one = card (ssingleton emptySet)
two :: Term
two = card (pair emptySet (ssingleton emptySet))
```

5.6. **Sums.** The value $f(x)$ corresponding to x of a function f, when G is the graph of f, is (slightly optimized) the y such that (x, y) is in G. Bourbaki defines (II §3.1, definition 3, remark 1) the image of a set X according to a graph G as

```
ens y (exists "x" (and (RIn (TVar "x") X)
                       (RIn (orderedPair (TVar "x") y) G)))
```

But since X is a singleton for our case, we don't need to check $x \in \{x\}$. I further simplify things and just say the value of x in G is that y such that $(x, y) \in G$.

```
val :: Term → Term → Term
val x graph = tau "y" (RIn (orderedPair x (TVar "y")) graph)
```

In a remark after Proposition 5 (III §3.3), Bourbaki notes if a and b are two cardinals, and I a set with two elements (e.g., the cardinal 2), then there exists a mapping f of I onto $\{a, b\}$ for which the sum of this family is denoted $a + b$.

The sum of a family of sets is discussed in (II §4.8) Definition 8 gives it as:

> Let $(X_i)_{i\in I}$ be a family of sets. The sum of this family is the union of the family of sets $(X_i \times \{i\})_{i\in I}$.

The notion of a family of sets is defined in (II §3.4). It is basically the graph of a function $I \to \{X_i\}$.

The union of a family of sets $(X_i)_{i\in I}$ is (II §4.1 Definition 1) $\mathcal{E}_x(\exists i)((i \in I)$ and $(x \in X))$ The family $\{X_0, X_1\}$ when $X_0 = X_1 = 1$ is then *cartesianProduct two one*. Combining this together, we get the sum as:

```
setSum :: Term → Term → Term
setSum idx family = ens "x" (exists "i"
                                (and (RIn (TVar "i") idx)
                                     (RIn (TVar "x") (val (TVar "i") family))))
```

When a and b are cardinal numbers, Bourbaki uses the indexed family $\{f_1, f_2\}$ where $f_1 = a$ and $f_2 = b$. This indexed family coincides with the cartesian product of the cardinality 2 with the unordered pair $\{a, b\}$. Then the sum of this family is the sum of cardinals.

```
cardSum :: Term → Term → Term
cardSum a b = setSum two (cartesianProduct two (pair a b))
```

Now, the big moment, the equation asserting $1 + 1 = 2$.

```
onePlusOneIsTwo :: Relation
onePlusOneIsTwo = REq two (cardSum one one)
```

5.7. **Curiousities.** I was curious about the length of various terms, so I defined them.

```
pairOfOnes :: Term
pairOfOnes = pair one one

productTwoOnes :: Term
productTwoOnes = cartesianProduct two pairOfOnes
```

6. Main Method

OK, ready? Your pulse is relaxed, you don't need a wet towel on your forehead or anything? Good, now we have the main method which will print out the statistics regarding the lengths of the various things:

```
main = do
  putStrLn ("The size of {x, y} = " ++ (show (len (pair (TVar "x") (TVar "y")))))
  putStrLn ("Size of (x, y) = " ++ (show (len (orderedPair (TVar "x") (TVar "y")))))
  putStrLn ("Size of the Empty Set = " ++ (show (len emptySet)))
  putStrLn ("Size of $X\\times Y$ = " ++ (show (len (cartesianProduct (TVar "X") (TVar "Y")))))
  putStrLn ("Size of 1 = " ++ (show (len one)))
  putStrLn ("Size of `{1,1}` = " ++ (show (len pairOfOnes)))
  putStrLn ("Size of `2*{1,1}` = " ++ (show (len productTwoOnes)))
  putStrLn ("Size of '1+1=2' = " ++ (show (len onePlusOneIsTwo)))
  putStrLn ("Size of 1* = " ++ (show (len (simp one))))
  putStrLn ("Size of A = " ++ (show (len (termA (ssingleton emptySet) "u" "U" "Z"))))
  putStrLn ("Size of B = " ++ (show (len (termB (ssingleton emptySet) "U" "Z"))))
  putStrLn ("Size of C = " ++ (show (len (termC (ssingleton emptySet) "U"))))
  putStrLn ("Size of D = " ++ (show (len (termD "U"))))
  putStrLn ("Size of E = " ++ (show (len (termE "U" "Z"))))
```

References

[1] Nicolas Bourbaki, *The Theory of Sets*. Springer, 2000 softcover reprint of 1968 English translation.

[2] Wayne Aitken, "Bourbaki, Theory of Sets, Chapter I, Description of Formal Mathematics: Summary and Commentary". Commentary dated 2022, https://public.csusm.edu/aitken_html/Essays/Bourbaki/BourbakiSetTheory1.pdf

URL: `https://github.com/pqnelson/bourbaki`