Formalizing Kelley's *Elementary Set Theory* in Naproche

1 Objectives of the Formalization Project

The appendix *Elementary Set Theory* to John L. Kelley's *General Topology* introduces the Kelley-Morse axiomatic theory of classes up to the axiom of choice and the beginnings of the theory of cardinal numbers. The text is written in a particularly formalistic style, with all notions explicitly introduced and defined in a mixture of natural language and symbolism. We shall try to approximate the text by a complete formalization in the Naproche proof system. The approximation concerns mathematical and logical structuring as well as typographical similarity.

Naproche supports the formalization and proof checking of texts which approximate the ordinary language and text structures used in mathematics. These texts are immediately readable by mathematicians without have to learn some formal language. On the other hand the texts are fully formal with respect to a formal grammar implemented in Naproche. There are a number of Naproche formalizations of short mathematical texts as well as some longer presentations of theories in set theory of linear algebra. Often, formalizations have to deviate considerably from rather informal or intuitive texts.

The Kelley text by its formalistic nature offers the possibility of a very close fit with the original. To achieve this aim involves work on several subproblems and subprojects:

- Basic understanding of set theoretical principles up to ordinals, the axiom of choice, and cardinals;
- Setting up development environments for Naproche (the Isabelle-Naproche PIDE) and for Haskell programming of Naproche, using stack and GHCi, or VSCode with Haskell extension;
- Organization through github;
- A "rough" formalization of the full Kelley text, in the current official version of Naproche (`https://files.sketis.net/Isabelle_Naproche-20190611/`);
- (Tackling complexity issues by breaking up the text into chapters;)
- Turning the input format into LaTeX;
- Enriching the ForTheL language by new words and constructs;
- ...
- Writing reports on the work;
- Preparing a publication to be submitted to some conference like CADE (Conference on Automated Deduction) or ITP (Interactive Theorem Proving) in 2021 or CICM / UITP

In these notes we present some work in these directions on an initial part of the Appendix. Starting from a Naproche formalization of the first 56 items of the Kelley text we describe steps to modify Naproche and the formalization to come closer to Kelley's original.

Table of contents

2 Kelley-Morse Class Theory

Georg Cantor characterized sets as follows:

> Unter einer *Menge* verstehen wir jede Zusammenfassung M von bestimmten, wohlunterschiedenen Objekten m unsrer Anschauung oder unseres Denkens (welche die "Elemente" von M genannt werden) zu einem Ganzen.

Such collections (Zusammenfassungen) are often denoted symbolically by

$$M = \{m | \, ...\}$$

where ... stands for some mathematical statement, usually involving the variable m.

Based on a naive understanding of such collections Cantor develops his set theory including a mathematical theory of the infinite. But contradictions can arise with some seemingly innocent collections like

$$M = \{m | \, m \notin m\}.$$

If we allow m to range over all mathematical notion, including M, then we get the contradiction

$$M \in M \leftrightarrow M \notin M.$$

This is the famous Russell contradiction or *paradox*.

So the formation of collections and their use have to be regulated, usually by axiomatic systems like Zermelo-Fraenkel set theory (ZF), von Neumann-Bernays-Gödel class theory (NBG) or Kelley-Morse class theory (KM).

The system KM consists of the following axioms, dispersed over the course of the appendix:

I Extensionality;

II Classification axiom-*scheme*, which says that every abstraction term is a class;

III Axiom of subsets ($\sim$ powerset);

IV Binary union;

V Substitution ($\sim$ replacement);

VI Amalgamation ($\sim$ general union);

VII Regularity (foundation);

VIII Infinity;

IX Axiom of Choice.

The system KM is slightly stronger than ZFC. Since mathematics can be conveniently formalized in ZFC, KM is also a good foundation for mathematics.

3 Sets in Naproche

The proof system Naproche provides mechanisms for forming and proving with collections. In Naproche, all abstraction terms are registered as `sets`. Since in KM all abstraction terms are classes, classes in KM correspond to sets in Naproche. We are using the Naproche distribution available at `https://files.sketis.net/Isabelle_Naproche-20190611/`
Indeed, the set formation mechanisms of Naproche readily allow to obtain the Russell contradiction. The following file is checked correctly by Naproche:

```
Definition. R = { set x | x is not an element of x}.

Lemma 1. R is a set.

Lemma 2. R is an element of R iff R is not an element of R.

Lemma 3. Contradiction.
```

In Naproche `set` is an inbuilt basic notion which is already part of the input language of Naproche which is called ForTheL for Formula Theory Language.
We cannot write `R = { x | x is not an element of x}.` because then the type of the variable `x` is not determined. Putting the notion `set` before the variable fixes its type. We could also write verbosely:

```
Definition. R is the set of sets x such that x is not an element of x.
```

These grammatical forms are defined in the module `SAD.ForTheL.Statement` of the Haskell source code. Reading or *parsing* of the input is carried out by (complex) parsers which are combinations of other, more basic parsers. The following parser definitions define abstraction terms:

```
set = label "set definition" $ symbSet <|> setOf
  where
    setOf = do
      wdTokenOf ["set", "sets"]; nm <- var -|- hidden; wdToken "of";
      (q, f, u) <- notion >>= single; vnm <- hidden
      vnmDecl <- makeDecl vnm;
      return (id, setForm vnmDecl $ subst (pVar vnm) (fst u) $ q f, [nm])
    symbSet = do
      (cnd, nm) <- symbSetNotation; h <- hidden
      nmDecl <- makeDecl nm
      return (id, setForm nmDecl $ cnd $ pVar nm, [h])
    setForm dcl = let nm = (Decl.name dcl, Decl.position dcl) in
      And (zSet zHole) . dAll dcl . Iff (zElem (pVar nm) zHole)


symbSetNotation = cndSet </> finSet
```

```
  where
    finSet = exbrc $ do
      ts <- sTerm `sepByLL1` smTokenOf ","; h <- hidden
      return (\tr -> foldr1 Or $ map (zEqu tr) ts, h)
    cndSet = exbrc $ do
      (tag, c, t) <- sepFrom; st <- smTokenOf "|" >> statement;
      vs <- freeVarPositions t; vsDecl <- mapM makeDecl vs;
      nm <- if isVar t then return $ (trName t, trPosition t) else hidden
      return (\tr -> tag $ c tr `blAnd` mbEqu vsDecl tr t st, nm)

    mbEqu _ tr Var{trName = v} = subst tr v
    mbEqu vs tr t = \st -> foldr mbdExi (st `And` zEqu tr t) vs
```

Let us comment on the red components:

- the parser `set` recognizes abstraction terms with `symbSet` or their verbose alternative with `setOf`;

- the parser combinator `<|>` forms an "or" of those two parsers;

- `setOf` expects one of the tokens "set" or "sets", then the token "of", and then some notion with a single variable;

- such notions can be rather complex, even involving "such that" subordinate clauses; in our case, e.g., `sets x such that x is not an element of x`;

- the parser `symbSet` is looking for a symbolic set notation by `symbSetNotation` and then wraps the result of that parser into some required format;

- `symbSetNotation` parser for abstraction terms with `cndSet` or for finite sets like $\{a, b, c\}$ with `finSet`;

- `cndSet` reads the outer braces {} with `exbrc`, looks for an abstraction variable with `sepFrom`, the symbolic token "|", and for the defining statement of the abstraction term with the parser `statement`;

- all (successful) parsing alternatives return a first-order formula using the funcion `setForm`; an important component is the clause `And (zSet zHole)` which registers the abstraction term or the verbose equivalent as a set through `zSet`.

Although it is hard to understand all details of these parser definitions, one can see some places where one can obviously modify the parsing. E.g., to get closer to Kelley's notation $\{x{:}\ldots\}$ for abstraction terms, one could substitute the parser `smTokenOf "|"` by `smTokenOf ":"`.

To come closer to the LaTeX-code for writing abstraction terms, one could require that in ASCII abstraction terms are written like `\{ set x | x is not an element of x \}`. This could be achieved by modifying the `exbrc` parser which reads outer braces. This parser is defined in the source module `SAD/Parser/Combinators` by:

```
-- mandatory parentheses, brackets, braces etc.
expar, exbrk, exbrc :: Parser st a -> Parser st a
expar p = snd <$> enclosed "(" ")" p
exbrk p = snd <$> enclosed "[" "]" p
exbrc p = snd <$> enclosed "{" "}" p
```

Remark 1. This motivates to define a variant

```
exbrc' p = snd <$> enclosed "\\{" "\\}" p
```

which would parse away the enclosing LaTeX braces \{ and \}; note that a single \ has a specific function in Haskell, so that the backslash character \ is denoted by \\. There are, however, some difficulties with tokenizing and parsing backslashes which will be addressed in a later section.

4 A First Formalization of Item 1 - 56

Our first Kelley formalization `Sections_1-56_ssets.ftl` deals with the appendix up to the definition of a relation and is done in the unmodified Naproche system. This means that abstraction terms are sets in Naproche and that classes which are sets in the sense of Kelley require a differently named notion. Here we use `sset` for "small sets". We shall later discuss ways of getting rid of these problems by modifying the Naproche code and the formalizations.

To work with Naproche, download the current Naproche-SAD package from

`https://files.sketis.net/Isabelle_Naproche-20190611--/`

We use the `Linux` version

`https://files.sketis.net/Isabelle_Naproche-20190611/Isabelle_Naproche-20190611_linux.tar.gz`

and unpack it somewhere in the file system. This creates a folder `Isabelle_Naproche-20190611` which contains an executable with the same name. Clicking or starting from a terminal opens `Isabelle-jedit` with `Naproche-SAD` available. One can now load `ForTheL` files, ending with `.ftl` and parse and check them.

Let us go through `Sections_1-56_ssets.ftl` and explain some aspects of the ForTheL language along the way.

4.1 Comments

ForTheL lines starting with a # are treating as comments:

```
# Kelley Morse Theory of Sets and Classes

# We formalize the

# Appendix: ELEMENTARY SET THEORY

# of John L. Kelley
# GENERAL TOPOLOGY
# D. Van Nostrand Company Inc. 1955
#
# The appendix develops what is known as Kelley-Morse
# class theory (KM).
# Kelley writes: "The system of axioms adopted is a variant
# of systems of Skolem and of A.P.Morse and owes much to
# the Hilbert-Bernays-von Neumann system as formulated
# by Gödel."

# This file covers the first 56 top level sections of the appendix.
# It uses SADs inbuilt class notion and mechanisms to model the
# classes of Kelley. We have built the class notion into
```

```
# Naproche-SAD by replacing "set" by "class"

# This file checks in ~1:30 min on my laptop.

# The Classification Axiom Scheme
```

4.2 Notions, Pretyping of Variables, and Synonyms

ForTheL is a typed language. Here, types are called notions, which serve to define the language at hand. Some notions like `sets` are built into the present Naproche. This allows to give a variable the type `set`, using a `Let ... stand for` command.

```
Let x, y, z, r, s, t, a, b, c, d, e stand for sets.
```

The `Let ... stand for` construct can also be used to introduce alternative notations for certain formulas. The relations of equality, inequality (`!=`) and elementhood (`is an element of`) are built-in. We want alternatives which are close to corresponding LaTeX code:

```
Let a \neq b stand for a != b.
Let a \in b stand for a is an element of b.
```

These are parser commands which synonymously exchange the new notation for the old notation whilst parsing.

4.3 Axioms

Global assumptions can be introduced by the `Axiom.` construct.

```
# Axiom I. Axiom of extent.
Axiom I. For each x for each y
x = y iff for each z z \in x iff z \in y.

# II Classification axiom-scheme corresponds to the way
# "classifications", i.e., abstraction terms are handled
# in Naproche-SAD
```

Axiom I consists of a single ForTheL statement. ForTheL provides a number of constructs known from ordinary mathematical writings. Alternatives are provided for extra flexibility. Instead of `For each x for each y ...` we could also write

```
For each x,y ...
For every x,y ...
For all x,y ...
```

Note that we cannot form the variable list as `x and y`, although this may be possible in later extensions of Naproche. The possible variants are described in the ForTheL handbook by a formal grammar. Ultimately they are defined by the parser definitions in the source code.

4.4 Definitions

```
[synonym sset/-s]
Definition 1. A sset is a set that is an element of some set.
```

```
# Elementary Algebra of Classes

Definition 2. x \cup y = { sset z | z \in x or z \in y }.

Definition 3. x \cap y = { sset u | u \in x and u \in y }.

Let the union of x and y stand for x \cup y.
Let the intersection of x and y stand for x \cap y.
```

Definition 1 introduces the new notion `sset` as a subnotion of `set` given by a defining formula (`that is ...`). Hovering the mouse above the Definition exhibits the first-order rendering of the definition:

```
[Translation] (line 41 of "/home/koepke/Desktop/Kelley_Formalization/
Sections_1-56_ssets.ftl")
forall v0 ((HeadTerm :: aSset(v0)) iff (aSet(v0) and exists v1 (aSet(v1)
and aElementOf(v0,v1))))
```

So the definition has introduced a new unary relation symbol `aSset` to the language and the defining equivalence will be part of the axiomatic assumptions. Note also that the "linguistic command" `[synonym sset/-s]` has registered `ssets` as a synonym for `sset`. This allows to achieve some grammatical correctness without using advanced linguistic methods. It is up to the user to use correct grammatical forms. Naproche itself does not recognize singulars or plurals.

Definitions 2 and 3 introduce new infix function symbols `\cup` and `\cap`:

```
[Translation] (line 45 of "/home/koepke/Desktop/Kelley_Formalization/
Sections_1-56_ssets.ftl")
(aSet(x) and aSet(y))
[Translation] (line 45 of "/home/koepke/Desktop/Kelley_Formalization/
Sections_1-56_ssets.ftl")
forall v0 ((HeadTerm :: v0 = x\cup y) iff (aSet(v0) and forall v1
(aElementOf(v1,v0) iff (Replacement :: (aSset(v1) and (aElementOf(v1,x)
or aElementOf(v1,y)))))))
```

Observe that the first Translation expressing the previous pretyping of `x` and `y`. In the defining formula, the use of the abstraction term has been eliminated using an equivalent first-order formula.

Finally (verbose) synonyms are introduced for these operations.

4.5 Theorems

We can now formulate our first theorems. The Isabelle environment shows that these theorems are accepted without giving explicit proofs. The reasoner of Naproche and the external automatic theorem prover *eprover* are able to construct proofs.

```
Theorem 4. (z \in x \cup y iff z \in x or z \in y)
and (z \in x \cap y iff z \in x and z \in y).

Theorem 5. x \cup x = x and x \cap x = x.

Theorem 6. x \cup y = y \cup x and x \cap y = y \cap x.
```

```
Theorem 7. (x \cup y) \cup z = x \cup (y \cup z)
and (x \cap y) \cap z = x \cap (y \cap z).

Theorem 8. x \cap (y \cup z) = (x \cap y) \cup (x \cap z)
and x \cup (y \cap z) = (x \cup y) \cap (x \cup z).
```

4.6 Synonyms Instead of Definitions

```
# 9 Definition, as a parser directive.
Let a \notin b stand for a is not an element of b.
```

Sometimes, using synonyms help to keep proof complexities down. A definition like

```
Definition 9. a \notin b iff a is not an element of b.
```

would introduce the definitional equivalence as a new premise, which enlarges the search space for proofs. The `Let ... stand for` replacement command is already evaluated during parsing.

4.7 The Universe of Small Sets

The class of all sets of KM becomes the set of all ssets in the present formalization. We also have some easy theorems.

```
Definition 18. UU = {sset x | x = x}.
Let the universe stand for UU.

Theorem 19. x \in UU iff x is an sset.

Theorem 20. x \cup UU = UU and x \cap UU = x.

Theorem 21. ~ 0 = UU and ~ UU = 0.
```

4.8 Big Intersections and Unions

In set theory one can interpret a class x as a family of sets and define

$$\bigcap x = \bigcap_{y \in x} y.$$

These "big" operations satisfy a number of laws:

```
Definition 22. \bigcap x = {sset z | for each y if y \in x then z \in y}.

Definition 23. \bigcup x = {sset z | for some y (z \in y and y \in x)}.

Let the intersection of x stand for \bigcap x.
Let the union of x stand for \bigcup x.

Theorem 24. \bigcap 0 = UU and \bigcup 0 = 0.
```

4.9 Subclasses

```
Definition 25. A subset of y is a set x such that each element of x is an
```

```
element of y. Let x \subset y stand for x is a subset of y.
Let x is contained in y stand for x \subset y.

Lemma. 0 \subset 0 and 0 \notin 0.

Theorem 26. 0 \subset x and x \subset UU.

Theorem 27. x = y iff x \subset y and y \subset x.

Theorem 28. If x \subset y and y \subset z then x \subset z.

Theorem 29. x \subset y iff x \cup y = y.

Theorem 30. x \subset y iff x \cap y = x.

Theorem 31. If x \subset y then \bigcup x \subset \bigcup y
and \bigcap y \subset \bigcap x.

Theorem 32. If x \in y then x \subset \bigcup y
and \bigcap y \subset x.
```

The subclass relation $\subset$ is the most important settheoretial relation after $\in$. However, Definition 25 does not define that relation directly, but it defines rather the *notion* of being a subset of a given set y: subset of y is a dependent type depending on the variable y. So one uses the construct for defining new notions.

4.10 The Axioms of Subsets and Separation

KM has an *Axiom of Subsets* which includes the standard ZF axioms of separation and power.

```
# Existence of sets

# Axiom of subsets.
Axiom III. If x is a sset then there is a sset y such that for each
z if z \subset x then z \in y.

# This axiom is a kind of powerclass axiom, where the powerclass
# also has all subCLASSES as elements.

Theorem 33. If x is an sset and z \subset x then z is an sset.

Theorem 34. 0 = \bigcap UU and UU = \bigcup UU.

Theorem 35. If x \neq 0 then \bigcap x is a sset.

Definition 36. 2^x = {sset y | y \subset x}.

Theorem 37. UU = 2^UU.
```

Theorem 33 is basically the axiom (schema) of separation: subclasses of sets are sets themselves.

4.11 Proving (Powerset)

So far, all theorem have been proved automatically by Naproche. Proving the powerset axiom requires a little proof for which ForTheL provides an environment of the form:

```
Proof.
 ...
qed.
```

We can now formulate:

```
Theorem 38. If x is a sset then 2^x is a sset and for
each y  y \subset x iff y \in 2^x.
Proof. Let x be a sset.
Take a sset y such that for each z
if z \subset x then z \in y (by III).
2^x \subset y.
qed.
```

The theorem claims an implication. A standard proof method is to assume the left-hand side and prove the right-hand side unter the extra assumption. We then take a witness `y` for Axiom III for the parameter `x`. ... Proofs have to end with `qed.` or `end.` Observe the closing fullstops!

4.12 The Russell Paradox

```
# The Russell paradox.

Definition. RR = {sset x | x \notin x}.

Theorem. RR is not a sset.

Theorem 39. UU is not a sset.
```

In this setup, we luckily do not get a contradiction, but we get that certain classes like the Russell class `RR` or the universe `UU` are too "big" to be ssets.

4.13 Parsing Finite Sets

A basic operation in set theory is the formation of singleton sets $\{x\}$, containing the single element x. This notation is already available in Naproche by the parser `finset` for *finite sets*:

```
symbSetNotation = cndSet </> finSet
  where
    finSet = exbrc $ do
      ts <- sTerm `sepByLL1` smTokenOf ","; h <- hidden
      return (\tr -> foldr1 Or $ map (zEqu tr) ts, h)
       ...
```

The parser accepts terms of the form $\{t_1, ..., t_n\}$ where the t_i are symbolic terms accepted by the parser `sterm`. We describe that parser in some detail. `finset`

- reads the outer braces $\{\}$, provided the list $t_1, ..., t_n$ is accepted by the parsing after the `$`-sign;

- the list is accepted by the parser `sepByLL1 sTerm (smTokenOf ",")`; `sepByLL1` is a parser combinator which alternatingly iterates the parsers `sTerm` and `smTokenOf ","` along the list until the end of the list;

- the terms found by `sTerm` are combined into a list `ts` of "parsed terms";

- `sepByLL1` can be viewed as a binary operation on parsers which can be written infix as `‘sepByLL1‘: sTerm ‘sepByLL1‘ smTokenOf ","`;

- `hidden` produces a new variable for further use;

- the parser `finSet` returns a function from terms to formulas which can be described intuitively as

$$\mathrm{tr} \mapsto \bigvee_{t \in \mathrm{ts}} \mathrm{tr} = t;$$

The formula is obtained by mapping the "equality making" function `zEqu tr` onto the list $[t_1, ..., t_n]$ which results in the list $[\mathrm{tr} = t_1, ..., \mathrm{tr} = t_n]$; then the `Or` operation is "folded" over that list by `foldr1` to yield the big disjunction

$$\mathrm{tr} = t_1 \vee (\mathrm{tr} = t_2 \vee (....));$$

- the returned function can be viewed as a formula with a "hole" or "slot" into which a further term could be inserted; the parse result of a formula

$$x \in \{t_1, ..., t_n\}$$

is obtained by applying that function to the variable x which yields the desired translation

$$\bigvee_{t \in \mathrm{ts}} x = t;$$

In this way the set notation has been reduced to a first-order formula without such notation.

- actually the parser returns an ordered pair consisting of the "substitution function" and the new "hidden" variable for further use.

4.14 Singletons in Kelley

We formalize Kelley's

> 40 Definition $\{x\} = \{z$: if $x \in \mathcal{U}$, then $z = y\}$.
> *Singleton* x is $\{x\}$.

as

```
Definition 40. <x> = {sset z | if x \in UU then z = x}.
Let the singleton of x stand for <x>.

# Before We used <x> instead of {x} since {x} was an inbuilt
# set notation.
```

Note that we use angle brackets $<...>$ instead of braces $\{\}$ to avoid conflicts with the inbuilt notation `{ . . . }`. Otherwise parsing `{x}` later would leed to an ambiguity because this string could be parsed by `finSet` as well as by the new parser created by `Definition 40`.

Another way to avoid such ambiguity and get further towards a LaTeX-format would be to use LaTeX-style `\{` and `\}` brackets:

```
Definition 40'. \{x\} = {sset z | if x \in UU then z = x}.
Let the singleton of x stand for \{x\}.
```

Note that `\{x\}` prettyprints as $\{x\}$. Further steps towards LaTeX could lead to an ASCII form of that definition like:

```
\begin{definition}[40]
$\{x\} = \{sset z : \text{ if } x \in \cal{U} \text{ then } z = x\}$.
Let the \em{singleton} of $x$x stand for $\{x\}$.
\end{definition}
```

One could (re-)define the definition environment to print out similar to Kelley's original:

> 40 DEFINITION $\{x\} = \{z\colon \text{ if } x \in \mathcal{U}, \text{ then } z = y\}$.
> *Singleton* x is $\{x\}$.

4.15 $\mathcal{U}$ as "Undefined"

We prove some properties of singletons:

```
Theorem 41. If x is a sset then for each y y \in <x> iff y = x.

Theorem 42. If x is a sset then <x> is a sset.
Proof. Let x be a sset. Then <x> \subset 2^x
and 2^x is a sset.
qed.

Theorem 43. <x> = UU iff x is not a sset.

Theorem 44a. If x is a sset then \bigcap <x> = x
and \bigcup <x> = x.
Theorem 44b. If x is not a sset then \bigcap <x> = 0
and \bigcup <x> = UU.
```

Note that we use `UU` as value of some terms which are not properly defined. This is a common convention in class theory.

4.16 Unions and Unordered Pairs

```
Axiom IV. If x is a sset and y is a sset then x \cup y is a sset.

Definition 45. <x,y> = <x> \cup <y>.
Let the unordered pair of x and y stand for <x,y>.

# The following has been a problem before:
# We use <x,y> instead of {x y} because Naproche-SAD requires
# some symbolic or textual material between the variables
# x and y. We use {x;y} instead of {x,y} because the latter
# notion is an inbuilt set notation of Naproche-SAD.

Theorem 46a. If x is a sset and y is a sset
then <x,y> is a sset and (z \in <x,y> iff z=x or z=y).
```

```
Theorem 46b. <x,y> = UU iff x is not a sset or y is not a sset.

Theorem 47a. If x,y are ssets then \bigcap <x,y> = x \cap y
and \bigcup <x,y> = x \cup y.
Proof. Let x,y be ssets.
\bigcup <x,y> \subset x \cup y.
x \cup y \subset \bigcup <x,y>.
qed.

Theorem 47b. If x is not a sset or y is not a sset then
\bigcap <x,y> = 0 and \bigcup <x,y> = UU.
```

Here is another problem why we do not (yet) use the inbuilt finite set notation. We want to form terms like $\{x\} \cup \{y\}$ or $\bigcup \{x, y\}$. In Naproche, for some efficiency reasons, set notation can only be used at the top-level of formulas but not within other terms. This restriction does not apply to user-defined function symbols.

4.17 Ordered Pairs

Ordered pairs (x, y) satisfy the characteristic property of `Theorem 55`:

$$(x, y) = (r, s) \to x = r \text{ and } y = s.$$

Ordered pairs can be understood as a basic logical construct without a need to "build" an ordered pair from other notions. In strict set theory, however, everything is required to be definable from sets (or classes). So one has to come up with a set-theoretic interpretation or implementation of ordered pairs. `Definition 48` uses the Kuratowski (-Wiener) construction:

$$(x, y) = \{\{x\}, \{x, y\}\}.$$

Since the notation (x, y) is already present in Naproche, we write $[x, y]$ instead.

Eventually the ordered pair should be taken as a basic notion satisfying `Theorem 55`. This would also simplify proof-checking, since a proof could not go astray by digging into the set structure of (x, y).

```
# Ordered Pairs; Relations

Definition 48. [x,y] = <<x>,<x,y>>.
Let the ordered pair of x and y stand for [x,y].

Theorem 49a. [x,y] is a sset iff x is a sset and y is a sset.
Theorem 49b. If [x,y] is not a sset then [x,y] = UU.

Theorem 50a. If x and y are ssets then
  \bigcup [x,y] = <x,y> and
  \bigcap [x,y] = <x> and
  \bigcup \bigcap [x,y] = x and
  \bigcap \bigcap [x,y] = x and
  \bigcup \bigcup [x,y] = x \cup y and
  \bigcap \bigcup [x,y] = x \cap y.

Theorem 50b. If x is not a sset or y is not a sset then
```

```
  \bigcup \bigcap [x,y] = 0 and
  \bigcap \bigcap [x,y] = UU and
  \bigcup \bigcup [x,y] = UU and
  \bigcap \bigcup [x,y] = 0.

Definition 51. 1st z = \bigcap \bigcap z.

Definition 52. 2nd z = (\bigcap \bigcup z) \cup
((\bigcup \bigcup z) ~ \bigcup \bigcap z).
Let the first coordinate of z stand for 1st z.
Let the second coordinate of z stand for 2nd z.

Theorem 53. 2nd UU = UU.

Theorem 54a. If x and y are ssets
then 1st [x,y] = x and 2nd [x,y] = y.
Proof. Let x and y be ssets.
2nd [x,y] = (\bigcap \bigcup [x,y]) \cup
((\bigcup \bigcup [x,y]) ~ \bigcup \bigcap [x,y])
= (x \cap y) \cup ((x \cup y) ~ x)
= y.
qed.

Theorem 54b. If x is not a sset or y is not a sset then
1st [x,y] = UU and 2nd [x,y] = UU.

Theorem 55. If x and y are ssets and [x,y] = [r,s] then
x = r and y = s.

# We can interpret UU to mean undefined.
# Then ( , ) produces a a set or undefined.
# We can instead extend the signature (our language)
# by an elementary symbol ( , ), satisfying standard axioms ... .
# Ideally, we would like ( , ) to an "object" and
# not a set. Sets will also be objects.

Definition 56. A relation is a set r such that for each element z of r
there exist x and y such that z = [x,y].
```

5 Programming Naproche

We describe how to run Naproche-SAD, change the source code, recompile it and run the new version of Naproche-SAD on the basis of the easily installed precompiled `tar.gz` distribution provided by Makarius Wenzel at `www.sketis.net`.
Download the current Naproche-SAD package from

```
https://files.sketis.net/Isabelle_Naproche-20190611--/
```

We use the `Linux` version

```
https://files.sketis.net/Isabelle_Naproche-20190611/Isabelle_Naproche-
20190611_linux.tar.gz
```

and unpack it somewhere in the file system. This creates a folder `Isabelle_Naproche-20190611` which contains an executable with the same name. Clicking or starting from a terminal opens `Isabelle-jedit` with `Naproche-SAD` available. One can now load `ForTheL` files, ending with `.ftl` and parse and check them.
Isabelle uses a `Naproche-SAD` binary at

```
. . . /Isabelle_Naproche-20190611/contrib/naproche-20190418/x86_64-linux
```

If that does not exist Isabelle searches the binary at

```
. . . /Isabelle_Naproche-20190611/contrib/naproche-20190418/.stack-work/
install/x86_64-linux/lts-12.25/8.4.4/bin
```

where it will be put by a `stack build` command issued in the folder

```
. . . /Isabelle_Naproche-20190611/contrib/naproche-20190418
```

We shall use that binary for our experiments. The first run of `stack build` compiles the main program at `.../contrib/naproche-20190418/app/Main.hs` with all submodules at `.../contrib/naproche-20190418/src/SAD`.
From the command line we can run the freshly compiled `Naproche-SAD` by a `stack exec` command; the command is `stack exec Naproche-SAD` and the input file to be checked is put behind two hashes `--`; also other input like options would be placed behind those `--`:

```
koepke@dell:~/TEST/Isabelle_Naproche-20190611/contrib/naproche-20190418$
stack exec Naproche-SAD -- examples/powerset.ftl
[Parser] "examples/powerset.ftl"
parsing successful
[Reasoner] "examples/powerset.ftl"
verification started
[Reasoner] "examples/powerset.ftl" (line 23, column 22)
goal:  Take a function f that is defined on M and surjects onto the
powerset of M.
[Reasoner] "examples/powerset.ftl" (line 24, column 1)
goal:  Define N = { x in M | x is not an element of f[x] }.
[Reasoner] "examples/powerset.ftl" (line 25, column 1)
goal:  Then N is not equal to the value of f at any element of M.
[Reasoner] "examples/powerset.ftl" (line 26, column 1)
goal:  Contradiction.
[Export] Error: Bad prover response:
# No SInE strategy applied
# Auto-Mode selected heuristic G_E___207_C18_F1_AE_CS_SP_PI_PS_S0S
# and selection function SelectComplexG.
#
# Presaturation interreduction done
# Proof found!
# SZS status ContradictoryAxioms
```

This is a successful check of the example file `powerset.ftl` which comes with the distribution and proves that the powerset of a set is strictly larger than the set itself.
To work with the binary built by `stack build` one can hide the preferred `Naproche-SAD` in

```
. . . /Isabelle_Naproche-20190611/contrib/naproche-20190418/x86_64-linux
```

by renaming it, e.g., to `Naproche-SAD.original` and open Isabelle by clicking on the icon in the folder `Isabelle_Naproche-20190611`.

We can change `Naproche-SAD` sources, recompile by `stack build` and restart Isabelle. Unfortunately, (re-)starting Isabelle_Naproche with its default settings can take a minute or so.

5.1 Example: Changing | to : in Abstraction Terms

We demonstrate the reprogramming of Naproche by introducing : as a separator in abstraction terms. The original | separator is defined in the module `SAD.ForTheL.Statement` in the definition of `cndSet`:

```
symbSetNotation = cndSet </> finSet
  where
    finSet = ...
    cndSet = exbrc $ do
      (tag, c, t) <- sepFrom; st <- smTokenOf "|" >> statement;
      vs <- freeVarPositions t; vsDecl <- mapM makeDecl vs;
      nm <- if isVar t then return $ (trName t, trPosition t) else hidden
      return (\tr -> tag $ c tr `blAnd` mbEqu vsDecl tr t st, nm)

    mbEqu _ tr Var{trName = v} = subst tr v
    mbEqu vs tr t = \st -> foldr mbdExi (st `And` zEqu tr t) vs
```

We only have to replace `smTokenOf "|"` by `smTokenOf ":"`.

This is organized as follows. We enumerate our experiments by 01, 02, 03, ... The original situation has the number 01. So we rename the original formalization file to `Sections_1-56_ssets01.ftl`. The original Naproche sources are put into a folder `.../contrib/naproche-20190418/src01`, and then we can put modified sources into `.../contrib/naproche-20190418/src`. The preferred `Naproche-SAD` binary in

`.../Isabelle_Naproche-20190611/contrib/naproche-20190418/x86_64-linux`

is renamed to `Naproche-SAD.original`.

After putting the (slightly) modified new sources into the folder `.../src` we open a terminal, go to the folder

`.../Isabelle_Naproche-20190611/contrib/naproche-20190418`

and issue the command `stack build` there. Compilation will take a while.

We then click the `Isabelle_Naproche-20190611` binary in the `.../Isabelle_Naproche-20190611` folder.

When we open `Sections_1-56_ssets01.ftl` we get an error in the first line with an abstraction term with a |-separator. Unfortunately the error message is not about an unexpected | , but an unexpected "." at the end of the sentence. We then go through the file, replacing all separators.

Thereafter the modified file checks correctly. We now save the modified file as `Sections_1-56_ssets02.ftl`.

Remark 2. If we want the alternative that abstraction terms could have : *or* | as separators, we could have used the alternative `smTokenOf ":" <|> smTokenOf "|"`:

```
symbSetNotation = cndSet </> finSet
  where
    finSet = ...
    cndSet = exbrc $ do
      (tag, c, t) <- sepFrom;
      st <- (smTokenOf ":" <|> smTokenOf "|") >> statement;
      ...
```

6 Classes and Sets in ForTheL

6.1 Substituting `set` by `class` and `sset` by `set`

Currently the word `"set"` is used in the input where we would like to see the word `"class"`. Therefore we go through the code and look for parsers accepting the word `"set"` or `"sets"`. We modify them to accept `"class"` or `"classes"` instead.

For this one should use a search tool that can search for words across folders. Such search is provided in VSCode or in the classic Midnight Commander (mc in a Linux terminal). `"set"` is found in two places:

- In `SAD/ForTheL/Base` the initial state of the ForTheL parser is defined. We see that the initial notions are `function`, `set`, `element of`, and `object`:

```
initFS = FState
  eq [] nt sn
  cf rf [] []
  [] [] [] sp
  [] [] []
  0 0 0 []
  where
    ...
    nt = [
      ([Wd ["function","functions"], Nm], zFun . head),
      ([Wd ["set","sets"], Nm], zSet . head),
      ([Wd ["element", "elements"], Nm, Wd ["of"], Vr], \(x:m:_) -> zElem
x m),
      ([Wd ["object", "objects"], Nm], zObj . head)]
```

We simply substitute `"set","sets"` by `"class", "classes"`.

- In `SAD/ForTheL/Statement` `"set"` is part of the parser `set` which accepts phrases like "set of ...":

```
set = label "set definition" $ symbSet <|> setOf
  where
    setOf = do
      wdTokenOf ["set", "sets"]; nm <- var -|- hidden; wdToken "of";
      ...
```

Again substitute `"set","sets"` by `"class", "classes"`.

Thereafter we recompile the code of Naproche and restart Isabelle-Naproche. The file `Sections_1-56_ssets02.ftl` has to be modified by replacing `"set"` by `"class"` in line with the software modification.

Now that the word `"set"` is no longer used, we can also replace the provisional `sset` by `set` in the formalization.

We replace the provisional definition

```
[synonym sset/-s]
Definition 1. A sset is a set that is an element of some set.
```

by

```
[synonym set/-s]
Definition 1. A set is a class that is an element of some class.
```

Then we replace `sset` by `set` throughout the document.
We save the new software as `src03` and the new formalization as `Section_1-56_03.ftl`.

Remark 3. Surprisingly the checking time for our file is now less than half of the previous checking time. Maybe this has to do with less ontological checking since we have changed the ontology?

6.2 Optional: Replacing `set` by `class` Throughout the Source Code

So far, the word "set" is still used in the source code, since so far we have only changed the input language. In the following we describe, how to eliminate those "sets" as well in favour of "classes". This would be the prefered option for cleanliness and consistency of code, put we do not implement it in the current experiment.
We go through the code and turn every "set" into "class". We do this by, e.g., saving the original file `SAD.Core.Base.hs` as `SAD.Core.Base.hs.old` and modifying `SAD.Core.Base.hs`.
In the module `SAD.Core.Base` replace `set` by `clss` (since `class` is a Haskell keyword) and `zSet` by `zClass`:

```
-- initial definitions
initialDefinitions = IM.fromList [
  (-1,  equality),
  (-2,  less),
  (-4,  function),
  (-5,  functionApplication),
  (-6,  domain),
  (-7,  clss),
  (-8,  elementOf),
  (-10, pair) ]

equality  = DE [] Top Signature (zEqu (zVar "?0") (zVar "?1")) [] []
less      = DE [] Top Signature (zLess (zVar "?0") (zVar "?1")) [] []
clss     = DE [] Top Signature (zClass $ zVar "?0") [] []
elementOf = DE [zClass $ zVar "?1"] Top Signature
  (zElem (zVar "?0") (zVar "?1")) [] [[zClass $ zVar "?1"]]
function  = DE [] Top Signature (zFun $ zVar "?0") [] []
domain    = DE [zFun $ zVar "?0"] (zClass ThisT) Signature
  (zDom $ zVar "?0") [zClass ThisT] [[zFun $ zVar "?0"]]
pair      = DE [] Top Signature (zPair (zVar "?0") (zVar "?1")) [] []
functionApplication =
  DE [zFun $ zVar "?0", zElem (zVar $ "?1") $ zDom $ zVar "?0"] Top
Signature
    (zApp (zVar "?0") (zVar "?1")) []
```

```
    [[zFun $ zVar "?0"],[zElem (zVar $ "?1") $ zDom $ zVar "?0"]]


initialGuards = foldr (\f -> DT.insert f True) (DT.empty) [
  zClass $ zVar "?1",
  zFun $ zVar "?0",
  zElem (zVar $ "?1") $ zDom $ zVar "?0"]
```

By these initial definitions, the notion of `class` is given the identifier `-7`. Furthermore `zSet` is renamed to `zClass`, and certain other notions include that variables in certain places are now classes, like in

```
elementOf = DE [zClass $ zVar "?1"] Top Signature
  (zElem (zVar "?0") (zVar "?1")) [] [[zClass $ zVar "?1"]]
```

The formula $x \in y$ includes and requires that y is a class.
Now the definition of `zSet` in the module has to become a definition of `zClass`. In `SAD.Data.Formula.Kit` we put classes instead of sets in

```
-- creation of predefined functions and notions
zEqu t s  = zTrm equalityId "=" [t,s]
zLess t s = zTrm lessId "iLess" [t,s]
zThesis   = zTrm thesisId "#TH#" []
zFun      = zTrm functionId "aFunction" . pure
zApp f v  = zTrm applicationId "sdtlbdtrb" [f , v]
zDom      = zTrm domainId "szDzozmlpdtrp" . pure
zClass    = zTrm classId "aClass" . pure
zElem x m = zTrm elementId "aElementOf" [x,m]
zProd m n = zTrm productId "szPzrzozdlpdtcmdtrp" [m, n]
zPair x y = zTrm pairId "slpdtcmdtrp" [x,y]
zObj      = zTrm objectId "aObj" . pure -- this is a dummy for parsing
purposes

-- predefined identifiers

equalityId    =  -1 :: Int
lessId        =  -2 :: Int
thesisId      =  -3 :: Int
functionId    =  -4 :: Int
applicationId =  -5 :: Int
domainId      =  -6 :: Int
classId       =  -7 :: Int
elementId     =  -8 :: Int
productId     =  -9 :: Int
pairId        = -10 :: Int
objectId      = -11 :: Int
```

We now have to replace `zSet` by `zClass`. In the initial parser state in the module `SAD.ForTheL.Base` we replace the original assignment of `zSet` to the word pattern for set:

```
([Wd ["set","sets"], Nm], zSet . head)
```

by

```
([Wd ["class","classes"], Nm], zClass . head)
```

In the module `SAD.ForTheL.Statement` there are many occurances of "sets". We have replaced all of them, at a certain risk of loosing some set mechanism for separation or replacement, that we shall have to reconstitute later. Once sets are reintroduced as a notion, something like `zSet` will have to be defined.

Finally `zSet` has to be renamed in the modules `SAD.Core.Reason` and `SAD.Core.ProofTask`.

There are also some set-related functions which have been renamed at the same time.

Now we compile the code with `stack build`. We realize from the errors that a few more renamings are necessary.

7 Integrating ForTheL Constructs into LaTeX

We experiment with some changes to the Naproche code and to the formalization (of the Kelley appendix) so that eventually ForTheL will be a sublanguage of LaTeX. Using dedicated style files and package files and possibly some writing conventions this should lead to a faithful prettyprinting of the mathematical content.

The present modifications concern specific issues of the Kelley formalization and give valuable information for a deep and systematic integration of ForTheL into LaTeX. Most of the following experimentyl changes will be ad hoc to attain naturality for the text at hand. E.g., at the moment we crudely eliminate all `$` and `\` from the logical processing, although `$` is an important "semantic" switch between text and mathematics and would help to distinguish the variable a from the word "a". Eliminating `\` also falsely identifies `\R` with `R`, although one might want to use them as two distinct identifiers.

A deeper understanding or reading of LaTeX input will be needed in the long term: whereas `\cal{U}` extends the alphabet by a symbol $\mathcal{U}$, `\bar{A}` can denote the value of A under some operation $A \mapsto \bar{A}$. There is also notation which can go both ways: $a', b', \ldots$ can be used as identifiers, but f' suggests a derivation operator applied to a function f.

In the end there will be architectural decisions between alternative uses, sometimes perhaps selectable by the user.

7.1 Preparing for LaTeX Input

We would like to put parts of the input file between `$`-signs to request math mode and mathematical typesetting. These `$`-signs momentarily play no role in the logical processing and so we already filter them out during tokenizing. (Eventually, math mode will also be used by the Naproche processing and will be dealt differently from the crude approach now.)

Tokenizing is carried out by the function `tokenize` in the module `SAD/Parser/Token`:

```
tokenize :: SourcePos -> String -> [Token]
tokenize start = posToken start False
  where
    posToken pos ws s
      | not (null lexem) =
          makeToken lexem pos ws True : posToken (advancesPos pos lexem)
False rest
      where (lexem, rest) = span isLexem s

    posToken pos _ s
      | not (null white) = posToken (advancesPos pos white) True rest
```

```
      where (white, rest) = span isSpace s

    posToken pos ws s@('#':_) =
      makeToken comment pos False False : posToken (advancesPos pos
comment) ws rest
      where (comment, rest) = break (== '\n') s

    posToken pos ws (c:cs) =
      makeToken [c] pos ws True : posToken (advancePos pos c) False cs

    posToken pos _ _ = [EOF pos]
```

The tokenizer scans the input and cuts it up into tokens, eliminating whitespace (which also includes newline commands). `tokenize` is defined by pattern matching including logical guards to organize the distinction between proper text and whitespace. Other (symbolic) characters are turned into "singleton" tokens by the second but last clause.

Since the pattern matching clauses are processed in the order of their position in the source code, we can capture and eliminate a `$` by putting a clause:

```
    posToken pos ws ('$':cs) =
      posToken (advancePos pos '$') False cs
```

before the "singleton" clause.

Since we are embedding ForTheL into LaTeX we can also exchange the ForTheL comment character `#` by the LaTeX comment character `%`.

The new tokenizer is:

```
tokenize :: SourcePos -> String -> [Token]
tokenize start = posToken start False
  where
    posToken pos ws s
      | not (null lexem) =
          makeToken lexem pos ws True : posToken (advancesPos pos lexem)
False rest
      where (lexem, rest) = span isLexem s

    posToken pos _ s
      | not (null white) = posToken (advancesPos pos white) True rest
      where (white, rest) = span isSpace s

    posToken pos ws s@('%':_) =
      makeToken comment pos False False : posToken (advancesPos pos
comment) ws rest
      where (comment, rest) = break (== '\n') s

    posToken pos ws ('$':cs) =
      posToken (advancePos pos '$') False cs

    posToken pos ws (c:cs) =
      makeToken [c] pos ws True : posToken (advancePos pos c) False cs
```

```
posToken pos _ _ = [EOF pos]
```

We install the new tokenizer and test it on the file `Section_1-56_03.ftl` which throws all kinds of errors.

- Starting up Isabelle-Naproche, there is an error in `init.opt`, since the configuration files of Naproche are also evaluated using the same tokenizer. `init.opt` starts with a comment line beginning with a `#`. So we simply delete that line. After that we have to restart Isabelle-Naproche.

- in `Section_1-56_03.ftl` we now have to replace all `#` by `%`, using a "replace all" tool.

Now our file, renamed `Section_1-56_04.ftl` checks. We can test whether it eliminates `$`-signs by inserting them somewhere in the file.

This works, and rechecking the altered file only requires reparsing. The internal representation of the text is not altered since the token sequence of the input is not altered. This reparsing takes place within a few seconds.

The original

```
% Axiom I. Axiom of extent.
Axiom I. For every x,y
x = y iff for each z z \in x iff z \in y.
```

can now be augmented for LaTeX as

```
% Axiom I. Axiom of extent.
Axiom I. For every $x,y$
$x = y$ iff for each $z$ $z \in x$ iff $z \in y$.
```

This piece prettyprints as:

> Axiom I. For every x, y $x = y$ iff for each z $z \in x$ iff $z \in y$.

getting closer to Kelley's original:

> I Axiom of extent *For each x and each y it is true that $x = y$ if and only if for each z, $z \in x$ when and only when $z \in y$.*

A better approximation is also possible in ForTheL by

```
% Axiom I. Axiom of extent.
Axiom I. For each $x,y$
$x = y$ if and only if for each $z$ $z \in x$ if and only if $z \in y$.
```

Obviously, we could also change the code to provide "it is true that" and a "when and only when" phrases. More tricky would be a "for each … and each …" quantificational construct.

Remark 4. Axiom I involves two "iffs", and one should check that these are parsed correctly, i.e., bracketed correctly in the first-order translation. Luckily (!?) this is the case here:

```
[Translation] (line 33 of "/home/koepke/Desktop/Kelley_Formalization/
Sections_1-56_04.ftl")
forall v0 forall v1 ((aSet(v0) and aSet(v1)) implies (v0 = v1 iff forall
v2 (aSet(v2) implies (aElementOf(v2,v0) iff aElementOf(v2,v1)))))
```

We insert many `$`'s for standard prettyprinting into `Section_1-56_04.ftl`.
To print out braces {....} in LaTeX requires to have

```
\{ ... \}
```

in the LaTeX-source. So we have to "escape" braces by \. This is easily done in the ForTheL text for the singleton and unordered pair notation. E.g., the original definition of unordered pair

```
Definition 45. <x,y> = <x> \cup <y>.
Let the unordered pair of x and y stand for <x,y>.
```

now reads

```
Definition 45. $\{x,y\} = \{x\} \cup \{y\}$.
Let the unordered pair of $x$ and $y$ stand for $\{x,y\}$.
```

and prints out as

> Definition 45. $\{x, y\} = \{x\} \cup \{y\}$. Let the unordered pair of x and y stand for $\{x, y\}$.

7.2 Tokenizing LaTeX

In LaTeX, the backslash \ is used as special character serving multiple purposes. Words starting with a backslash like `\cup` are commands for printing special symbols like ∪; generally, LaTeX commands start with a backslash. Moreover, backslashes are used for some specific printing functions like

- `\\` for line breaks;
- `\ ` to insert specific horizontal spaces;
- `\{` and `\}` for LaTeX braces (to break out from the structuring uses of `{` and `}` in LaTeX documents);
- `\begin{...}` and `\end{...}` of LaTeX environments;
- ...

The multiple uses of backslashes pose difficulties for the current parsing of ForTheL texts: `...\cup...` and `...\{...` are tokenized as `[...,\,cup,...]` and `[...,\,{,...]` resp. When, e.g., the `statement` parser is confronted with the token list for the end of an abstraction term, say, `[\,{,...]` it will consume the \ and expect a token like `cup`; on seeing a `{` the `statement` parser will fail; but the \ is already consumed so that a parser for `\}` will also fail.
To avoid this problem, `\cup` should be tokenized into a single token `\cup` whereas `\{` could be handled as one or two tokens. Also one could extend the tokenizer to treat `\\` and `\ ` with a blank as whitespace.

7.2.1 Running the Tokenizer

So far we have only run the Haskell program Naproche-SAD using `stack exec Naproche-SAD ...` from a command line or in the Isabelle-Naproche framework. Another way to run Haskell modules is via the interactive version of the Glasgow Haskell Compiler GHCi. Opening ghci with the desired module in a standard terminal leads to errors:

```
koepke@dell:~/Desktop/Kelley_Formalization/Isabelle_Naproche-20190611/
contrib/naproche-20190418$ ghci src/SAD/Parser/Token.hs
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling SAD.Parser.Token ( src/SAD/Parser/Token.hs,
interpreted )

src/SAD/Parser/Token.hs:25:1: error:
    Could not find module ‘SAD.Core.SourcePos’
    Use -v to see a list of the files searched for.
   |
25 | import SAD.Core.SourcePos
   | ^^^^^^^^^^^^^^^^^^^^^^^^^

src/SAD/Parser/Token.hs:26:1: error:
    Could not find module ‘SAD.Core.Message’
...
...
    Failed, no modules loaded.
```

Obviously the system could not find certain modules because search paths were not available. The `stack` environment for Haskell is able to provide those paths. So we prefix `ghci` with `stack`:

```
koepke@dell:~/Desktop/Kelley_Formalization/Isabelle_Naproche-20190611/
contrib/naproche-20190418$ stack ghci src/SAD/Parser/Token.hs
Using configuration for Naproche-SAD:lib to load /home/koepke/Desktop/
Kelley_Formalization/Isabelle_Naproche-20190611/contrib/naproche-
20190418/src/SAD/Parser/Token.hs
Configuring GHCi with the following packages: Naproche-SAD
GHCi, version 8.4.4: http://www.haskell.org/ghc/  :? for help
[ 1 of 13] Compiling Isabelle.Naproche ( /home/koepke/Desktop/
Kelley_Formalization/Isabelle_Naproche-20190611/contrib/naproche-
20190418/Isabelle/src/Isabelle/Naproche.hs, interpreted )
...
...
[13 of 13] Compiling SAD.Parser.Token ( /home/koepke/Desktop/
Kelley_Formalization/Isabelle_Naproche-20190611/contrib/naproche-
20190418/src/SAD/Parser/Token.hs, interpreted )
Ok, 13 modules loaded.
Loaded GHCi configuration from /tmp/haskell-stack-ghci/abbfd0d6/ghci-
script
*SAD.Parser.Token>
```

At the `ghci` prompt `*SAD.Parser.Token>` one can now apply the function `tokenize` from the module `SAD.Parser.Token` to various inputs. We first determine the type of the function by using the `ghci` command `:t`

```
*SAD.Parser.Token> :t tokenize
tokenize :: SourcePos -> String -> [Token]
```

More information is provided by the `:i` or `:info` command which also specifies the location of the definition of the function:

```
*SAD.Parser.Token> :i tokenize
tokenize :: SourcePos -> String -> [Token]
        -- Defined at /home/koepke/Desktop/Kelley_Formalization/
Isabelle_Naproche-20190611/contrib/naproche-20190418/src/SAD/Parser/
Token.hs:65:1
```

We `tokenize` with an "empty" source position:

```
*SAD.Parser.Token> tokenize noPos "Hello World"
[Hello,World,]
```

Note 5. For studying tokenizing in Naproche we use the original module `SAD.Parser.Token` which does *not* eliminate the `$`-sign.

Tokenizing breaks the text into "words", eliminating whitespace. Internally, tokens have several data fields.

```
data Token =
  Token {
    tokenText :: String,
    tokenPos :: SourcePos,
    tokenWhiteSpace :: Bool,
    tokenProper :: Bool} |
  EOF {tokenPos :: SourcePos}
```

The show function for tokens only displays their string content in the field `tokenText`.

```
showToken :: Token -> String
showToken t@Token{} = tokenText t
showToken EOF{} = "end of input"
```

Let us see how the important LaTeX symbols `\,{,},$,_,^,%` are tokenized. We tokenize a LaTeX source for the following "theorem":

Theorem 6. $a_0^2 + a_1^2 = a_2^2$.

```
*SAD.Parser.Token> tokenize noPos "\\begin{theorem} $a_0^2 + a_1^2 =
a_2^2$. \\}. \\end{theorem}"
[\,begin,{,theorem,},$,a_0,^,2,+,a_1,^,2,=,a_2,^,2,$,.,\,},.,\,end,{,
theorem,},]
```

Note that when Haskell reads a backslash in an input file, it is always replaced by a double backslash `\\`, because the single `\` is used as an escape or control character in strings; a newline in a string is, e.g., represented as `\n`.
We see that all of `\,{,},$,_,^` are treated as separate tokens, except that the underscore `_` is treated as part of the alphabet. To adapt the tokenizing to Naproche, the following changes should be made:

- To support the use of LaTeX commands like `\cup` this should be read as one token. On reading a letter or a `\` the tokenizer should continue reading *letters* (*not* numbers) as long as possible, and the result of the reading should then be put into one ordinary token.

– The underscore _ should be treated like any other symbol and yield a single token.

– The ForTheL comment sign # should be exchanged for the LaTeX comment sign %.

– Experiments have shown, that LaTeX commands like \{ and \} for displaying braces should be turned into *one* token.

– At the moment we would eliminate all $-sign during tokenizing, so that prettyprinting and logical processing are somewhat "orthogonal". (Eventually, the tokenizing should distinguish between text mode and math mode and derive semantic information from that.)

– Otherwise the default tokenizing if other cases fail would ensure that single characters become single tokens.

There are some other issues that one should attend to later:

– Beyond the current ForTheL whitespace, LaTeX whitespace like \\ for newline or \ for extra space should eventually be tokenized away as whitespace. (For the moment we could however write texts only using ForTheL whitespace.)

The code for making a token out of an ordinary word is

```
tokenize :: SourcePos -> String -> [Token]
tokenize start = posToken start False
  where
    posToken pos ws s
      | not (null lexem) =
          makeToken lexem pos ws True : posToken (advancesPos pos lexem)
False rest
      where (lexem, rest) = span isLexem s
     ...
```

For LaTeX commands starting with \ we can insert a similar second clause in the definition of `tokenize` to obtain the new tokenizer

```
tokenize :: SourcePos -> String -> [Token]
tokenize start = posToken start False
  where
    posToken pos ws s
      | not (null lexem) =
          makeToken lexem pos ws True : posToken (advancesPos pos lexem)
False rest
      where (lexem, rest) = span isLexem s

    posToken pos ws ('\\':s)
      | not (null alpha) =
          makeToken ('\\':alpha) pos ws True : posToken (advancesPos pos
('\\':alpha)) False rest
      where (alpha, rest) = span isAlpha s

    posToken pos ws ('\\':'{':s) =
      makeToken "\\{" pos ws True : posToken (advancesPos pos "\\{")
False s
```

```
    posToken pos ws ('\\':'}':s) =
      makeToken "\\}" pos ws True : posToken (advancesPos pos "\\}")
False s

    posToken pos _ s
      | not (null white) = posToken (advancesPos pos white) True rest
      where (white, rest) = span isSpace s

    posToken pos ws s@('%':_) =
      makeToken comment pos False False : posToken (advancesPos pos
comment) ws rest
      where (comment, rest) = break (== '\n') s

    posToken pos ws ('$':cs) =
      posToken (advancePos pos '$') False cs

    posToken pos ws (c:cs) =
      makeToken [c] pos ws True : posToken (advancePos pos c) False cs

    posToken pos _ _ = [EOF pos]
tokenize :: SourcePos -> String -> [Token]

isLexem :: Char -> Bool
isLexem c = isAscii c && isAlphaNum c
```

After compiling we test the tokenizer with

```
*SAD.Parser.Token> tokenize noPos "\\begin{theorem} It is true that \n
\\n $\\{set v_0 | v_0 \\neq v_1\\} = \\emptyset$.\\end{theorem}"
[\begin,{,theorem,},It,is,true,that,\n,\{,set,v,_,0,|,v,_,0,\neq,v,_,1,
\},=,\emptyset,.,\end,{,theorem,},]
*SAD.Parser.Token>
*SAD.Parser.Token> tokenize noPos "\\alpha\\alpha"
[\alpha,\alpha,]
*SAD.Parser.Token> tokenize noPos "\\alpha123\\"
[\alpha,123,\,]
*SAD.Parser.Token> tokenize noPos "\\LaTeX"
[\LaTeX,]
*SAD.Parser.Token> tokenize noPos "alpha1 12miles alpha_1"
[alpha1,12miles,alpha,_,1,]
*SAD.Parser.Token>
```

So the tokenizer respects LaTeX commands, as well as ForTheL parsing of words that do not contain \ or _. We can now restart `Isabelle_Naproche`. A line with an abstraction like

```
Definition 2. x cup y =  \{ set z : z in x or z in y \}.
```

is now parsed correctly with translation

```
[Translation] (line 45 of "...")
(aSet(x) and aSet(y))
```

```
[Translation] (line 45 of " . . . ")
forall v0 ((HeadTerm :: v0 = xcup y) iff (aSet(v0) and forall v1
(aElementOf(v1,v0) iff (Replacement :: (aSet(v1) and (aElementOf(v1,x) or
aElementOf(v1,y)))))))
```

We save this version of Naproche as `src05`. It is not worthwhile to get the Kelley formalization accepted by this version, since momentarily `\in` or `\cup` or not accepted.

7.3 A Shortcut to LaTeX Prettyprinting

In this subsection we describe a "dirty" approach to prettyprinting. We are already tokenizing away the $-signs, and we extend this also to the backslash \. This means that we only view $ and \ as printing commands that have nothing, or hardly anything to do with the logical processing of the text. Thus we view `\alpha` as the ForTheL variable `alpha` where the backslash is a "tag" which only indicates that this variable could be printed out as α provided that we are "between $-signs".
Under this approach, the

```
Lemma. $\alpha$ = alpha.
```

is a tautology, which would prettyprint as

Lemma. α=alpha.

To avoid such oddities one should adopt the convention or style to use the prettyprinting version of an identifier *everywhere* if its prettyprinting is used somewhere.
The following tokenizer in `src06` implements the approach of erasing $ and \ signs before the logical processing:

```
tokenize :: SourcePos -> String -> [Token]
tokenize start = posToken start False
  where
    -- put LaTeX comments from % to end of line into a "False token"
    posToken pos ws s@('%':_) =
      makeToken comment pos False False : posToken (advancesPos pos
comment) ws rest
      where (comment, rest) = break (== '\n') s

    -- make alphanumeric tokens
    posToken pos ws s
      | not (null lexem) =
          makeToken lexem pos ws True : posToken (advancesPos pos lexem)
False rest
      where (lexem, rest) = span isLexem s

    -- make alphabetic tokens from LaTeX commands
    posToken pos ws ('\\':s)
      | not (null alpha) =
          makeToken alpha pos ws True : posToken (advancesPos pos
('\\':alpha)) False rest
      where (alpha, rest) = span isAlpha s

    -- drop LaTeX "\\" line breaks
```

```
    posToken pos ws ('\\':'\\':s) =
      posToken (advancesPos pos "\\\\") False s

    -- drop LaTeX "\ " spaces
    posToken pos ws ('\\':' ':s) =
      posToken (advancesPos pos "\\ ") False s

    -- drop whitespace
    posToken pos _ s
      | not (null white) = posToken (advancesPos pos white) True rest
      where (white, rest) = span isSpace s

    -- eliminate $-signs
    posToken pos ws ('$':cs) =
      posToken (advancePos pos '$') False cs

    -- eliminate other \-signs
    posToken pos ws ('\\':cs) =
      posToken (advancePos pos '\\') False cs

    -- put character being read into singleton token
    posToken pos ws (c:cs) =
      makeToken [c] pos ws True : posToken (advancePos pos c) False cs

    -- otherwise EOF
    posToken pos _ _ = [EOF pos]

isLexem :: Char -> Bool
isLexem c = isAscii c && isAlphaNum c
```

We can now parse *and* proofcheck the file `Section_1-56_05.ftl`. At the same time, the file is a piece of LaTeX code (without a preamble and `\begin{document}` or `\end{document}`) which prints out as

> Let $x, y, z, r, s, t, a, b, c, d, e$ stand for classes.
>
> Let $a \neq b$ stand for a != b. Let $a \in b$ stand for a is an element of b.
>
> Axiom I. For each x, y $x = y$ if and only if for each z $z \in x$ if and only if $z \in y$.
>
> [synonym set/-s]
>
> Definition 1. A set is a class that is an element of some class.
>
> Definition 2. $x \cup y = \{\text{set } z\colon z \in x \text{ or } z \in y\}$.
>
> Definition 3. $x \cap y = \{\text{set } u\colon u \in x \text{ and } u \in y\}$.
>
> Let the union of x and y stand for $x \cup y$. Let the intersection of x and y stand for $x \cap y$.
>
> Theorem 4. ($z \in x \cup y$ iff $z \in x$ or $z \in y$) and ($z \in x \cap y$ iff $z \in x$ and $z \in y$).
>
> Theorem 5. $x \cup x = x$ and $x \cap x = x$.
>
> Theorem 6. $x \cup y = y \cup x$ and $x \cap y = y \cap x$.
>
> Theorem 7. $(x \cup y) \cup z = x \cup (y \cup z)$ and $(x \cap y) \cap z = x \cap (y \cap z)$.
>
> Theorem 8. $x \cap (y \cup z) = (x \cap y) \cup (x \cap z)$ and $x \cup (y \cap z) = (x \cup y) \cap (x \cup z)$.
>
> Let $a \notin b$ stand for a is not an element of b.

Definition 10. $\sim x = \{\text{set } y\colon y \notin x\}$. Let the complement of x stand for $\sim x$.

Theorem 11. $\sim(\sim x) = x$.

Theorem 12. $\sim(x \cup y) = (\sim x) \cap (\sim y)$ and $\sim(x \cap y) = (\sim x) \cup (\sim y)$.

Definition 13. $x \sim y = x \cap (\sim y)$.

Theorem 14. $x \cap (y \sim z) = (x \cap y) \sim z$.

Definition 15. $0 = \{\text{set } x\colon x \neq x\}$. Let the void class stand for 0. Let zero stand for 0.

Theorem 16. $x \notin 0$.

Theorem 17. $0 \cup x = x$ and $0 \cap x = 0$.

Definition 18. $UU = \{\text{set } x\colon x = x\}$. Let the universe stand for UU.

Theorem 19. $x \in UU$ iff x is a set.

Theorem 20. $x \cup UU = UU$ and $x \cap UU = x$.

Theorem 21. $\sim 0 = UU$ and $\sim UU = 0$.

Definition 22. $\bigcap x = \{\text{set } z\colon \text{for each } y \text{ if } y \in x \text{ then } z \in y\}$.

Definition 23. $\bigcup x = \{\text{set } z\colon \text{for some } y\ (z \in y \text{ and } y \in x)\}$.

Let the intersection of x stand for $\bigcap x$. Let the union of x stand for $\bigcup x$.

Theorem 24. $\bigcap 0 = UU$ and $\bigcup 0 = 0$.

Definition 25. A subclass of y is a class x such that each element of x is an element of y. Let $x \subset y$ stand for x is a subclass of y. Let x is contained in y stand for $x \subset y$.

Lemma. $0 \subset 0$ and $0 \notin 0$.

Theorem 26. $0 \subset x$ and $x \subset UU$.

Theorem 27. $x = y$ iff $x \subset y$ and $y \subset x$.

Theorem 28. If $x \subset y$ and $y \subset z$ then $x \subset z$.

Theorem 29. $x \subset y$ iff $x \cup y = y$.

Theorem 30. $x \subset y$ iff $x \cap y = x$.

Theorem 31. If $x \subset y$ then $\bigcup x \subset \bigcup y$ and $\bigcap y \subset \bigcap x$.

Theorem 32. If $x \in y$ then $x \subset \bigcup y$ and $\bigcap y \subset x$.

Axiom III. If x is a set then there is a set y such that for each z if $z \subset x$ then $z \in y$.

Theorem 33. If x is a set and $z \subset x$ then z is a set.

Theorem 34. $0 = \bigcap UU$ and $UU = \bigcup UU$.

Theorem 35. If $x \neq 0$ then $\bigcap x$ is a set.

Definition 36. $2^x = \{\text{set } y\colon y \subset x\}$.

Theorem 37. $UU = 2^U U$.

Theorem 38. If x is a set then 2^x is a set and for each y $y \subset x$ iff $y \in 2^x$.
Proof. Let x be a set. Take a set y such that for each z if $z \subset x$ then $z \in y$ (by III). $2^x \subset y$. qed.

Definition. $R\,R = \{\text{set } x\colon x \notin x\}$.

Theorem. $R\,R$ is not a set.

Theorem 39. UU is not a set.

Definition 40. $x = \{\text{set } z\colon \text{if } x \in UU \text{ then } z = x\}$. Let the singleton of x stand for x.

Theorem 41. If x is a set then for each y $y \in \{x\}$ iff $y = x$.

Theorem 42. If x is a set then $\{x\}$ is a set.

Proof. Let x be a set. Then $\{x\} \subset 2^x$ and 2^x is a set. qed.

Theorem 43. $\{x\} = UU$ iff x is not a set.

Theorem 44a. If x is a set then $\bigcap \{x\} = x$ and $\bigcup \{x\} = x$.

Theorem 44b. If x is not a set then $\bigcap \{x\} = 0$ and $\bigcup \{x\} = UU$.

Axiom IV. If x is a set and y is a set then $x \cup y$ is a set.

Definition 45. $\{x, y\} = \{x\} \cup \{y\}$. Let the unordered pair of x and y stand for $\{x, y\}$.

Theorem 46a. If x is a set and y is a set then $\{x, y\}$ is a set and ($z \in \{x, y\}$ iff $z = x$ or $z = y$).

Theorem 46b. $\{x, y\} = UU$ iff x is not a set or y is not a set.

Theorem 47a. If x, y are sets then $\bigcap \{x, y\} = x \cap y$ and $\bigcup \{x, y\} = x \cup y$. Proof. Let x, y be sets. $\bigcup \{x, y\} \subset x \cup y$. $x \cup y \subset \bigcup \{x, y\}$. qed.

Theorem 47b. If x is not a set or y is not a set then $\bigcap \{x, y\} = 0$ and $\bigcup \{x, y\} = UU$.

Definition 48. $[x, y] = \{\{x\}, \{x, y\}\}$. Let the ordered pair of x and y stand for $[x, y]$.

Theorem 49a. $[x, y]$ is a set iff x is a set and y is a set.

Theorem 49b. If $[x, y]$ is not a set then $[x, y] = UU$.

Theorem 50a. If x and y are sets then $\bigcup [x, y] = \{x, y\}$ and $\bigcap [x, y] = \{x\}$ and $\bigcup \bigcap [x, y] = x$ and $\bigcap \bigcap [x, y] = x$ and $\bigcup \bigcup [x, y] = x \cup y$ and $\bigcap \bigcup [x, y] = x \cap y$.

Theorem 50b. If x is not a set or y is not a set then $\bigcup \bigcap [x, y] = 0$ and $\bigcap \bigcap [x, y] = UU$ and $\bigcup \bigcup [x, y] = UU$ and $\bigcap \bigcup [x, y] = 0$.

Definition 51. 1st $z = \bigcap \bigcap z$.

Definition 52. 2nd $z = (\bigcap \bigcup z) \cup ((\bigcup \bigcup z) \sim \bigcup \bigcap z)$. Let the first coordinate of z stand for 1st z. Let the second coordinate of z stand for 2nd z.

Theorem 53. 2nd $UU = UU$.

Theorem 54a. If x and y are sets then 1st $[x, y] = x$ and 2nd $[x, y] = y$. Proof. Let x and y be sets. 2nd $[x, y] = (\bigcap \bigcup [x, y]) \cup ((\bigcup \bigcup [x, y]) \sim \bigcup \bigcap [x, y]) = (x \cap y) \cup ((x \cup y) \sim x) = y$. qed.

Theorem 54b. If x is not a set or y is not a set then 1st $[x, y] = UU$ and 2nd $[x, y] = UU$.

Theorem 55. If x and y are sets and $[x, y] = [r, s]$ then $x = r$ and $y = s$.

Definition 56. A relation is a class r such that for each element z of r there exist x and y such that $z = [x, y]$.

Note that we have made some small adjustments to the text beyond \$'s and \'s, like replacing `~` by `\sim` since `~` is a LaTeX spacing command. We save the current Naproche source as `src06` and the formalization as `Sections_1-56_06.ftl`.

Obviously skipping all `$` and `\` is a trick that is not really adequate to the semantic or logical content that is also present in the LaTeX code. One example is the treatment of variables. Keeping track of variables is a major issue in Naproche. Sometimes words are read as variables although they should be part of the natural language text. If we make a distinction between text- and math-mode through `$` signs or other, such ambiguities could be handled.

To better integrate ForTheL into LaTeX and leveraging the extra information transmitted by the typesetting is part of the further development of Naproche.

7.4 LaTeX Environments

A typical feature of LaTeX are `\begin{ . . . } . . . \end{ . . . }` environments which trigger some specific typesetting. We also want to use these "brackets" for structuring ForTheL texts. As an example we shall fuse together the ForTheL `Definition` environment with LaTeX's `\begin{definition} . . . \end{definition}`.

Then ForTheL environment is a `topsection` of the ForTheL text defined in `SAD.ForTheL.Structure`:

```
topsection = signature <|> definition <|> axiom <|> theorem

definition =
  let define = pretype $ pretypeSentence Posit defExtend defVars noLink
  in  genericTopsection Definition defH define
```

In a ForTheL definition like

> Definition 56. A relation is a class r such that for each element z of r there exist x and y such that $z = [x, y]$.

we shall change `Definition 56.` to `\begin{definition}[56]` and we shall adjoin a `\end{definition}` after the final dot.

Let us first consider the situation without the numbering variable (`56`). The header of a definition is defined as

```
defH = header ["definition"]
```

with

```
header titles = finish $ markupTokenOf topsectionHeader titles >> optLL1
"" topIdentifier
```

If there are no top identifiers between `Definition` and the subsequent fullstop "`.`" then `defH` only parses for "`Definition.`". If we ignore the markup aspect then `defH` checks for `Definition` and the fullstop and issues the empty string `""`.

So we could replace `defH` by

```
defH = wdToken "begin" >> smTokenOf "{" >> wdToken "definition" >>
wdToken "}" >> return ""
```

Instead, to cater for the markup, we let `markupTokenOf` look for `begin` which generates the mark `topsectionHeader`:

```
defH = markupTokenOf topsectionHeader ["begin"]
         >> smTokenOf "{"
         >> wdToken "definition"
         >> wdToken "}"
         >> return ""
```

This is used within the general topsection construct

```
genericTopsection kind header endparser = do
  pos <- getPos; inp <- getInput; nm <- header;
  toks <- getTokens inp; bs <- body
  let bl = Block.makeBlock zHole bs kind nm [] pos toks
  addBlockReports bl; return bl
  where
    body = assumption <|> endparser
    assumption = topAssume `pretypeBefore` body
    topAssume = pretypeSentence Assumption (asmH >> statement) assumeVars
noLink
```

Among other things, `genericTopsection` in case of a definition:

- reads the header
- reads a sequence of assumption statements like "Let ..." or "Assume ..."
- uses `endparser` to read the actual definition

`endparser` is also responsible for ultimately ending the definition with a fullstop. For `definition` the endparser is the locally defined

```
define = pretype $ pretypeSentence Posit defExtend defVars noLink
```

We chase the definitions of the components for a fullstop and find

```
noLink = finish $ return []
```

where `finish` is defined in `SAD.Parser.Combinators`:

```
finish :: Parser st a -> Parser st a
finish p = after p dot
```

So we replace the `noLink` in `definition` by a parser that checks for `.\end{definition}`, tokenized:

```
endDef = smTokenOf "."
           >> wdToken "end"
           >> smTokenOf "{"
           >> wdToken "definition"
           >> smTokenOf "}"
           >> return []

definition =
  let define = pretype $ pretypeSentence Posit defExtend defVars endDef
  in  genericTopsection Definition defH define
```

With this modification the (unparametrized) definition environment is parsed. We modify the formalization accordingly and save it as `Sections_1-56_07.ftl`. This file prettyprints to

> Let $x, y, z, r, s, t, a, b, c, d, e$ stand for classes.
> Let $a \neq b$ stand for a != b. Let $a \in b$ stand for a is an element of b.

Axiom I. For each x, y $x = y$ if and only if for each z $z \in x$ if and only if $z \in y$.
[synonym set/-s]

Definition 7. *A set is a class that is an element of some class.*

Definition 8. $x \cup y = \{set\ z\colon z \in x\ \ or\ z \in y\}$.

Definition 9. $x \cap y = \{set\ u\colon u \in x\ \ and\ u \in y\}$.

Let the union of x and y stand for $x \cup y$. Let the intersection of x and y stand for $x \cap y$.
Theorem 4. $(z \in x \cup y$ iff $z \in x$ or $z \in y)$ and $(z \in x \cap y$ iff $z \in x$ and $z \in y)$.
Theorem 5. $x \cup x = x$ and $x \cap x = x$.
Theorem 6. $x \cup y = y \cup x$ and $x \cap y = y \cap x$.
Theorem 7. $(x \cup y) \cup z = x \cup (y \cup z)$ and $(x \cap y) \cap z = x \cap (y \cap z)$.
Theorem 8. $x \cap (y \cup z) = (x \cap y) \cup (x \cap z)$ and $x \cup (y \cap z) = (x \cup y) \cap (x \cup z)$.
Let $a \notin b$ stand for a is not an element of b.

Definition 10. $\sim x = \{set\ y\colon y \notin x\}$.

Let the complement of x stand for $\sim x$.
...
...

We now also change the signature, axiom and theorem environments. The relevant part of `SAD.ForThe.Structure` now reads:

```
-- topsections

signature =
  let sigext = pretype $ pretypeSentence Posit sigExtend defVars
endEnvironment
  in  genericTopsection Signature sigH sigext
    where sigH = markupTokenOf topsectionHeader ["begin"]
            >> smTokenOf "{"
            >> wdToken "signature"
            >> wdToken "}"
            >> return ""
          endEnvironment = smTokenOf "."
            >> wdToken "end"
            >> smTokenOf "{"
            >> wdToken "signature"
            >> smTokenOf "}"
            >> return []

definition =
  let define = pretype $ pretypeSentence Posit defExtend defVars
endEnvironment
  in  genericTopsection Definition defH define
    where defH = markupTokenOf topsectionHeader ["begin"]
```

```
            >> smTokenOf "{"
            >> wdToken "definition"
            >> wdToken "}"
            >> return ""
          endEnvironment = smTokenOf "."
            >> wdToken "end"
            >> smTokenOf "{"
            >> wdToken "definition"
            >> smTokenOf "}"
            >> return []
axiom =
  let posit = pretype $ pretypeSentence Posit (affH >> statement)
affirmVars endEnvironment
  in  genericTopsection Axiom axmH posit
    where axmH = markupTokenOf topsectionHeader ["begin"]
            >> smTokenOf "{"
            >> wdToken "axiom"
            >> wdToken "}"
            >> return ""
          endEnvironment = smTokenOf "."
            >> wdToken "end"
            >> smTokenOf "{"
            >> wdToken "axiom"
            >> smTokenOf "}"
            >> return []
theorem =
  let topAffirm = pretypeSentence Affirmation (affH >> statement)
affirmVars endEnvironment
  in  genericTopsection Theorem thmH (topProof topAffirm)
    where thmH = markupTokenOf topsectionHeader ["begin"]
            >> smTokenOf "{"
            >> wdTokenOf ["theorem", "lemma", "corollary", "proposition"]
            >> wdToken "}"
            >> return ""
          endEnvironment = smTokenOf "."
            >> wdToken "end"
            >> smTokenOf "{"
            >> wdTokenOf ["theorem", "lemma", "corollary", "proposition"]
            >> smTokenOf "}"
            >> return []
```

The prettyprinting of the formalization is

...

...

Theorem 11. *If x is a set and $z \subset x$ then z is a set.*

Theorem 12. *$0 = \bigcap UU$ and $UU = \bigcup UU$.*

Theorem 13. *If $x \neq 0$ then $\bigcap x$ is a set.*

Definition 14. $2^x = \{set\ y\colon y \subset x\}$.

Theorem 15. $UU = 2^U U$.

Theorem 16. *If x is a set then 2^x is a set and for each y $y \subset x$ iff $y \in 2^x$.*

Proof. Let x be a set. Take a set y such that for each z if $z \subset x$ then $z \in y$ (by III). $2^x \subset y$. qed.

Definition 17. $RR = \{set\ x\colon x \notin x\}$.

Theorem 18. *RR is not a set.*

Theorem 19. *UU is not a set.*

Definition 20. $\{x\} = \{set\ z\colon if\ x \in UU\ then\ z = x\}$.

Let the singleton of x stand for x.

Theorem 21. *If x is a set then for each y $y \in \{x\}$ iff $y = x$.*

Theorem 22. *If x is a set then $\{x\}$ is a set.*

Proof. Let x be a set. Then $\{x\} \subset 2^x$ and 2^x is a set. qed.

Theorem 23. *$\{x\} = UU$ iff x is not a set.*

...
...

We save the current Naproche source as `src07` and the formalization as `Sections_1-56_07.ftl`.

Remark 24. Problems: Parametrization, we can start a theorem and end with a lemma (agreement?),

7.5 The LaTeX Proof Environment

In ForTheL, proofs are subordinated to the statement to be proved. In a ForTheL theorem, this is the statement after the assumptions of the theorem, and this is read by the "end-parser" of `genericTopsection`. Parsing `\end{theorem} \begin{proof}` happens in the *middle* of the `endparser` of `genericTopSection`, which is equal to `(topProof topAffirm)` in the case of a `theorem`.

```
topProof p = do
  pre <- preMethod; bl <- p; post <- postMethod; typeBlock <- pretyping
bl;
  let pretyped = Block.declaredNames typeBlock
  nbl <- addDecl pretyped $ fmap TextBlock $ do
    nf <- indThesis (Block.formula bl) pre post
    addBody pre post $ bl {Block.formula = nf}
```

```
    return $ if null pretyped then [nbl] else [TextBlock typeBlock, nbl]
```

`topAffirm` is defined locally in `theorem`

```
let topAffirm = pretypeSentence Affirmation (affH >> statement)
affirmVars link
```

Going through the ramified definition of `theorem` we see that it parses text like

```
-- thmH in theorem:
\begin{theorem}
-- body in genericTopsection, using asmH, statement, and finish:
{(let us | we can) (assume | presume | suppose) [that] | let) statement.}
-- preMethod in topProof:
[[let us | we can] (prove | show | demonstrate)
[by (contradiction | case analysis | induction [on sTerm])] that]
-- lowIdentifier, affH, statement in statementBlock, used in topAffirm:
["(" topIdentifier ")"][then | hence | thus | therefore] statement
-- link in statementBlock:
[(by identifiers)] .
-- postMethod in topProof, . due to finish in explicit in postmethod:
[indeed | proof [by (contradiction |
      case analysis | induction [on sTerm]) . ]
-- further text depends on the declared "method" situation. In the
-- "explicit" situation with and explicit "proof ... .":
proofText, ending with
(qed | end | trivial | obvious).
-- the final dot comes from link in proofBody
```

with standard notation for optional and alternative phrasing.

The LaTeX `\begin{theorem}` has already been installed. `\end{theorem}` corresponds to the read dot `.` `\begin{proof}` corresponds to `proof.` and `\end{proof}` to `qed.`

We modify `SAD.ForTheL.Structure` to use these LaTeX environments. We have also modified the code for `Signature` but there are no signature extensions in our formalization. The resulting LaTeX now reads:

...

...

Theorem 25. $UU = 2^U U$.

Theorem 26. *If x is a set then 2^x is a set and for each y $y \subset x$ iff $y \in 2^x$.*

Proof. Let x be a set. Take a set y such that for each z if $z \subset x$ then $z \in y$. $2^x \subset y$. □

Definition 27. $RR = \{set\ x : x \notin x\}$.

Theorem 28. *RR is not a set.*

...

...

We save the new source as `src08` and the formalization as `Section_1-56_08.ftl`.

7.6 Text Structurings and Mathematical Comments

Obviously there is more to mathematical texts than just their logical skeleton. Texts also motivate and explain, and to convey such messages they have natural text structures. We want to display headings from Kelley like

```
\section{ELEMENTARY ALGEBRA OF CLASSES}
```

in LaTeX but eliminate it from the logical processing. Our provisional solution is to treat `\section` like a comment character in the tokenizer and put the text in brackets into a comment token. This can be achieved by augmenting `tokenize` by

```
    -- drop \section commands
     posToken pos ws s@('\\':'s':'e':'c':'t':'i':'o':'n':'{':_) =
       makeToken sec pos False False : posToken (advancesPos pos ('a':sec)
ws (tail rest)
       where (sec, rest) = break (== '}') s
```

Then we can add sections to the Formalization.

...

Axiom 29. *For each x, y $x = y$ if and only if for each z $z \in x$ if and only if $z \in y$.*

[synonym set/-s]

Definition 30. *A set is a class that is an element of some class.*

8 ELEMENTARY ALGEBRA OF CLASSES

Definition 31. $x \cup y = \{set\ z : z \in x\ or\ z \in y\}$.

Definition 32. $x \cap y = \{set\ u : u \in x\ and\ u \in y\}$.

...

8.1 Typographical Modifiers

Mathematicians use modified letters like $\mathcal{U}$, $\mathbf{A}$, or $\mathbb{R}$ as variables or identifiers. We can directly use these notations in definitions:

```
\begin{definition} $\cal{U} = \{$set $x : x = x\}$.\end{definition}
Let the universe stand for $\cal{U}$.
```

which prints as

Definition 33. $\mathcal{U} = \{set\ x : x = x\}$.

Let the universe stand for $\mathcal{U}$.

Note that we would have a different reading, if the letter `U` had been declared before as a variable. Then parsing the token list `["cal","{","U","}"]` would yield a pattern with a hole at the U-position. Parsing for new notions by

```
newNotion = do
  (n, u) <- newNtnPattern nvr;
  f <- MS.get >>= addExpr n n True
  return (f, u)
```

in `SAD.ForTheL.Extension` involves a subparser in `SAD.ForTheL.Pattern`:

```
nvr = do
  v <- var; dvs <- getDecl; tvs <- MS.gets tvrExpr
  guard $ fst v `elem` dvs || any (elem (fst v) . fst) tvs
  return $ pVar v
```

This parser parses for variables (by `var`) which have been "declared" or "typed" before.

8.2 Typographical Functions

We define the projections of an ordered pair by:

```
\begin{definition} $1^{st}$ coord $z = \bigcap \bigcap
z$.\end{definition}

\begin{definition} $2^{nd}$ coord $z = (\bigcap \bigcup z) \cup
((\bigcup \bigcup z) \sim \bigcup \bigcap z)$.\end{definition}
Let the first coordinate of $z$ stand for $1^{st}$ coord $z$.
Let the second coordinate of $z$ stand for$2^{nd}$ coord $z$.
```

which typesets as

Definition 34. 1^{st} *coord* $z = \bigcap \bigcap z$.

Definition 35. 2^{nd} *coord* $z = (\bigcap \bigcup z) \cup ((\bigcup \bigcup z) \sim \bigcup \bigcap z)$.

Let the first coordinate of z stand for 1^{st} coord z. Let the second coordinate of z stand for2^{nd} coord z.

Here the definition `$1^{st}$ coord $z` contains the previously introduced variable `z` , so that we define a unary function symbol in the argument `z`.

8.3 Adding Linguistic Phrases

Kelley uses the phrase "when and only when" besides "if and only if". We add this to `SAD.ForTheL.Base` :

```
iff = wdToken "iff" <|> mapM_ wdToken ["if", "and", "only", "if"]
        <|> mapM_ wdToken ["when", "and", "only", "when"]
```

using the alternative combinator for parsers. At the same place we also allow the phrase "there are" by

```
there = wdToken "there" >> wdTokenOf ["is","are","exists","exist"]
```

To introduce the phrase "it is true that" as syntactic sugar as used in Kelley is a bit more complicated since there is some interference with the already implemented phrase "it is wrong that" in `SAD.ForTheL.Statement`:

```
headed = quStatem <|> ifThenStatem <|> (wrongStatem </> trueStatem)
  where
    quStatem = liftM2 ($) quChain statement
    ifThenStatem = liftM2 Imp
      (markupToken Reports.ifThen "if" >> statement)
      (markupToken Reports.ifThen "then" >> statement)
    wrongStatem =
      mapM_ wdToken ["it", "is", "wrong", "that"] >> fmap Not statement
    trueStatem =
      mapM_ wdToken ["it", "is", "true", "that"] >> statement
```

Note that we connect the two alternative parsers `wrongStatem` and `trueStatem` by the combinator

```
------ Choose with lookahead
{-# INLINE (</>) #-}
(</>) :: Parser st a -> Parser st a -> Parser st a
(</>) f g = try f <|> g
```

in `SAD.Parser.Combinators`. `wrongStatem </> trueStatem` means that `wrongStatem` is tried, and that in case of failure `trueStatem` is started from *the same* token list. Parsing "it is true that" begins with `wrongStatem` eating tokens until failure when it encounters "true"; then the token list is reset to `[‘‘it’’, ‘‘is’’, ‘‘true’’, that’’, . . . ]` and `trueStatem` will now succeed (and continue to parse the statement behind the phrase).

Remark 36. If one wants to forbid odd repetitions of these kinds of phrases ("it is false that it is true that $x \neq x$" one should moreover use some "optional" parser combinator.

8.4 Sets and Classes, Again

So far we parse the word "class" into the internal first-order symbol `aSet`. Initially this can be understood as a harmless renaming of `aSet` at the ForTheL level. However, in the first definition of the text, we define "set" at the ForTheL level:

Definition 37. *A set is a class x such that for some y $x \in y$.*

This definition creates a new(?) internal predicate `aSet` which probably conflicts with the existing `aSet`.

Remark 38. Apparently Naproche does not check whether a symbol defined by a signature extension or a definition already exists in the system as demonstrated by a simple example:

Signature 39. *A number is a notion.*

Signature 40. *A point is a notion.*

Signature 41. 0 *is a number.*

Signature 42. 0 *is a point.*

Lemma 43. 0=0.

The parser accepts the text with the competing signature specifications of 0 and only complains about ambiguity in the Lemma, since it cannot figure out "which" 0 is meant. There should be another check for signatures and definitions to rule out such redefinitions.

Also the definition generates an unintended formula for sets which is added to the general premises:

```
[Translation] (line 45 of "/home/koepke/Desktop/Kelley_Formalization/
temp1.ftl")
forall v0 ((HeadTerm :: aSet(v0)) iff (aSet(v0) and exists v1 (aSet(v1)
and aElementOf(v0,v1))))
```

which is not intended to be introduced by a definition: if v_0 is a set then v_0 is an element of some (other) set. (Since the definition also identifies sets and classes this probably leads to the Russell contradiction.)
To avoid this, we rename the predicate for classes to `aClass` in `SAD.Data.Formula.Kit:196`:

```
zSet      = zTrm setId "aClass" . pure
```

Now the translation of the above definition is the intended definitional equivalence

```
[Translation] (line 45 of "/home/koepke/Desktop/Kelley_Formalization/
Sections_1-56_11.ftl")
forall v0 ((HeadTerm :: aSet(v0)) iff (aClass(v0) and exists v1
(aClass(v1) and aElementOf(v0,v1))))
```

We save the new source as `src08` and the formalization as `Section_1-56_08.ftl`.

9 THE CLASSIFICATION AXIOM SCHEME

Let $x, y, z, r, s, t, u, v, a, b, c, d, e$ stand for classes.
Let $a \neq b$ stand for a != b. Let $a \in b$ stand for a is an element of b.

Axiom 44. *For each x, y it is true that $x = y$ if and only if for each z $z \in x$ when and only when $z \in y$.*

[synonym set/-s]

Definition 45. *A set is a class x such that for some y $x \in y$.*

10 ELEMENTARY ALGEBRA OF CLASSES

Definition 46. $x \cup y = \{set\ z\colon z \in x\ \ or\ z \in y\}$.

Definition 47. $x \cap y = \{set\ u\colon u \in x\ and\ u \in y\}$.

Let the union of x and y stand for $x \cup y$. Let the intersection of x and y stand for $x \cap y$.

Theorem 48. $(z \in x \cup y\ iff\ z \in x\ or\ z \in y)\ and\ (z \in x \cap y\ iff\ z \in x\ and\ z \in y)$.

Theorem 49. $x \cup x = x\ and\ x \cap x = x$.

Theorem 50. $x \cup y = y \cup x\ and\ x \cap y = y \cap x$.

Theorem 51. $(x \cup y) \cup z = x \cup (y \cup z)\ and\ (x \cap y) \cap z = x \cap (y \cap z)$.

Theorem 52. $x \cap (y \cup z) = (x \cap y) \cup (x \cap z)\ and\ x \cup (y \cap z) = (x \cup y) \cap (x \cup z)$.

Let $a \notin b$ stand for a is not an element of b.

Definition 53. $\sim x = \{set\ y\colon y \notin x\}$.

Let the complement of x stand for $\sim x$.

Theorem 54. $\sim(\sim x) = x$.

Theorem 55. $\sim(x \cup y) = (\sim x) \cap (\sim y)\ and\ \sim(x \cap y) = (\sim x) \cup (\sim y)$.

Definition 56. $x \sim y = x \cap (\sim y)$.

Theorem 57. $x \cap (y \sim z) = (x \cap y) \sim z$.

Definition 58. $0 = \{set\ x\colon x \neq x\}$.

Let the void class stand for 0. Let zero stand for 0.

Theorem 59. $x \notin 0$.

Theorem 60. $0 \cup x = x\ and\ 0 \cap x = 0$.

Definition 61. $\mathcal{U} = \{set\ x\colon x = x\}$.

Let the universe stand for $\mathcal{U}$.

Theorem 62. $x \in \mathcal{U}$ *if and only if* x *is a set.*

Theorem 63. $x \cup \mathcal{U} = \mathcal{U}\ and\ x \cap \mathcal{U} = x$.

Theorem 64. $\sim 0 = \mathcal{U}\ and\ \sim \mathcal{U} = 0$.

Definition 65. $\bigcap x = \{set\ z\text{: }for\ each\ y\ if\ y \in x\ then\ z \in y\}$.

Definition 66. $\bigcup x = \{set\ z\text{: }for\ some\ y\ (z \in y\ and\ y \in x)\}$.

Let the intersection of x stand for $\bigcap x$. Let the union of x stand for $\bigcup x$.

Theorem 67. $\bigcap 0 = \mathcal{U}$ *and* $\bigcup 0 = 0$.

Definition 68. *A subclass of y is a class x such that for each z if $z \in x$ then $z \in y$.*

Let $x \subset y$ stand for x is a subclass of y. Let x is contained in y stand for $x \subset y$.

Lemma 69. $0 \subset 0$ *and* $0 \notin 0$.

Theorem 70. $0 \subset x$ *and* $x \subset \mathcal{U}$.

Theorem 71. $x = y$ *iff* $x \subset y$ *and* $y \subset x$.

Theorem 72. *If* $x \subset y$ *and* $y \subset z$ *then* $x \subset z$.

Theorem 73. $x \subset y$ *iff* $x \cup y = y$.

Theorem 74. $x \subset y$ *iff* $x \cap y = x$.

Theorem 75. *If* $x \subset y$ *then* $\bigcup x \subset \bigcup y$ *and* $\bigcap y \subset \bigcap x$.

Theorem 76. *If* $x \in y$ *then* $x \subset \bigcup y$ *and* $\bigcap y \subset x$.

11 EXISTENCE OF SETS

Axiom 77. *If x is a set then there is a set y such that for each z if $z \subset x$ then $z \in y$.*

Theorem 78. *If x is a set and $z \subset x$ then z is a set.*

Theorem 79. $0 = \bigcap \mathcal{U}$ *and* $\mathcal{U} = \bigcup \mathcal{U}$.

Theorem 80. *If* $x \neq 0$ *then* $\bigcap x$ *is a set.*

Definition 81. $2^x = \{set\ y\colon y \subset x\}$.

Theorem 82. $\mathcal{U} = 2^{\mathcal{U}}$.

Theorem 83. *If x is a set then 2^x is a set and for each y $y \subset x$ iff $y \in 2^x$.*

Proof. Let x be a set. Take a set y such that for each z if $z \subset x$ then $z \in y$. $2^x \subset y$. □

Definition 84. $\mathcal{R} = \{set\ x\colon x \notin x\}$.

Theorem 85. $\mathcal{R}$ *is not a set.*

Theorem 86. $\mathcal{U}$ *is not a set.*

Definition 87. $\{x\} = \{set\ z\colon if\ x \in \mathcal{U}\ then\ z = x\}$.

Let the singleton of x stand for x.

Theorem 88. *If x is a set then for each y $y \in \{x\}$ iff $y = x$.*

Theorem 89. *If x is a set then $\{x\}$ is a set.*

Proof. Let x be a set. Then $\{x\} \subset 2^x$ and 2^x is a set. □

Theorem 90. *$\{x\} = \mathcal{U}$ if and only if x is not a set.*

Theorem 91. *If x is a set then $\bigcap \{x\} = x$ and $\bigcup \{x\} = x$.*

Theorem 92. *If x is not a set then $\bigcap \{x\} = 0$ and $\bigcup \{x\} = \mathcal{U}$.*

Axiom 93. *If x is a set and y is a set then $x \cup y$ is a set.*

Definition 94. $\{x, y\} = \{x\} \cup \{y\}$.

Let the unordered pair of x and y stand for $\{x, y\}$.

Theorem 95. *If x is a set and y is a set then $\{x, y\}$ is a set and ($z \in \{x, y\}$ iff $z = x$ or $z = y$).*

Theorem 96. *$\{x, y\} = \mathcal{U}$ if and only if x is not a set or y is not a set.*

Theorem 97. *If x, y are sets then $\bigcap \{x, y\} = x \cap y$ and $\bigcup \{x, y\} = x \cup y$.*

Proof. Let x, y be sets. $\bigcup \{x, y\} \subset x \cup y$. $x \cup y \subset \bigcup \{x, y\}$. □

Theorem 98. *If x is not a set or y is not a set then $\bigcap \{x, y\} = 0$ and $\bigcup \{x, y\} = \mathcal{U}$.*

12 ORDERED PAIRS: RELATIONS

Definition 99. $[x, y] = \{\{x\}, \{x, y\}\}$.

Let the ordered pair of x and y stand for $[x, y]$.

Theorem 100. *$[x, y]$ is a set if and only if x is a set and y is a set.*

Theorem 101. *If $[x, y]$ is not a set then $[x, y] = \mathcal{U}$.*

Theorem 102. *If x and y are sets then $\bigcup [x, y] = \{x, y\}$ and $\bigcap [x, y] = \{x\}$ and $\bigcup \bigcap [x, y] = x$ and $\bigcap \bigcap [x, y] = x$ and $\bigcup \bigcup [x, y] = x \cup y$ and $\bigcap \bigcup [x, y] = x \cap y$.*

Theorem 103. *If x is not a set or y is not a set then $\bigcup \bigcap [x, y] = 0$ and $\bigcap \bigcap [x, y] = \mathcal{U}$ and $\bigcup \bigcup [x, y] = \mathcal{U}$ and $\bigcap \bigcup [x, y] = 0$.*

Definition 104. *1^{st} coord $z = \bigcap \bigcap z$.*

Definition 105. *2^{nd} coord $z = (\bigcap \bigcup z) \cup ((\bigcup \bigcup z) \sim \bigcup \bigcap z)$.*

Let the first coordinate of z stand for 1^{st} coord z. Let the second coordinate of z stand for2^{nd} coord z.

Theorem 106. *2^{nd} coord $\mathcal{U} = \mathcal{U}$.*

Theorem 107. *If x and y are sets then 1^{st} coord $[x, y] = x$ and 2^{nd} coord $[x, y] = y$.*

Proof. Let x and y be sets. 2^{nd} coord $[x, y] = (\bigcap \bigcup [x, y]) \cup ((\bigcup \bigcup [x, y]) \sim \bigcup \bigcap [x, y]) = (x \cap y) \cup ((x \cup y) \sim x) = y$. □

Theorem 108. *If x is not a set or y is not a set then 1^{st} coord $[x, y] = \mathcal{U}$ and 2^{nd} coord $[x, y] = \mathcal{U}$.*

Theorem 109. *If x and y are sets and $[x, y] = [u, v]$ then $x = u$ and $y = v$.*

Definition 110. *A relation is a class r such that for each element z of r there is x and y such that $z = [x, y]$.*

13 Changing the Format of Toplevel Definitions

Kelley defines the notion of "set" by:

1 DEFINITION *x is a set iff for some y, $x \in y$.*

The original ForTheL format for definitions requires to put the new notion first:

```
\begin{definition}
A set is a class $x$ such that for some $y$ $x \in y$.
\end{definition}
```

with prettyprinting

Definition 111. *A set is a class x such that for some y $x \in y$.*

The definition statement "A set is a class x such that for some y $x \in y$" is parsed by the parser

```
defNotion = do
  ((n,h),u) <- wellFormedCheck (ntnVars . fst) defn; uDecl <- makeDecl u
  return $ dAll uDecl $ Iff (Tag HeadTerm n) h
  where
    defn = do
      (n, u) <- newNotion; isOrEq; (q, f) <- anotion
      let v = pVar u; fn = replace v (trm n)
      h <- (fn . q) <$> dig f [v]
      return ((n,h),u)

    isOrEq = wdToken "=" <|> isEq
    isEq   = is >> optLL1 () (wdToken "equal" >> wdToken "to")
    trm Trm {trName = "=", trArgs = [_,t]} = t; trm t = t
```

which returns the defining formula $\mathrm{aSet}(v_0) \leftrightarrow (\mathrm{aClass}(v_0) \wedge \exists v_1(\mathrm{aClass}(v_1) \wedge v_0 \in v_1))$:

```
[Translation] (line 45 of "/home/koepke/Desktop/Kelley_Formalization/
Sections_1-56_13.ftl")
forall v0 ((HeadTerm :: aSet(v0)) iff (aClass(v0) and exists v1
(aClass(v1) and aElementOf(v0,v1))))
```

The line

```
(n, u) <- newNotion; isOrEq; (q, f) <- anotion
```

means that one first parses the pattern for the new notion, then for the keyword "is", then for the defining old notion, which in definitions is modified by a "such then" statement. In Kelley's formulation

1 DEFINITION *x is a set iff for some y, $x \in y$.*

the order of new and old notion is reversed, and the role of the "such then ..." is taken by an "iff ..." statement. It is however not sufficient to switch around the one line of the parser code. The statement after "iff" has to modify the old notion, and also some type casting is required.

The following code for the "endparser" `defExtend` of a definition can check the previous definition format by `defPredicate` and `defNotion` and the new format by `defNotion'`:

```
defExtend = defPredicat -|- defNotion' -|- defNotion
sigExtend = sigPredicat -|- sigNotion

defPredicat = ...

defNotion = do
  ((n,h),u) <- wellFormedCheck (ntnVars . fst) defn; uDecl <- makeDecl u
  return $ dAll uDecl $ Iff (Tag HeadTerm n) h
  where
    defn = do
      (n, u) <- newNotion; isOrEq; (q, f) <- anotion
      let v = pVar u; fn = replace v (trm n)
      h <- (fn . q) <$> dig f [v]
      return ((n,h),u)

    isOrEq = wdToken "=" <|> isEq
    isEq   = is >> optLL1 () (wdToken "equal" >> wdToken "to")
    trm Trm {trName = "=", trArgs = [_,t]} = t; trm t = t

defNotion' = do
  ((n,h),u) <- wellFormedCheck (ntnVars . fst) defn; uDecl <- makeDecl u
  return $ dAll uDecl $ Iff (Tag HeadTerm n) h
  where
    defn = do
      (q,f,w) <- (art >> gnotion basentn rat <|> fmap digadd primTvr) >>=
single
      is
      (n, u) <- newNotion
      iff
      s <- fmap (Tag Dig) statement
      let f' = And f s
      let g = subst zHole (fst w) f'
      let v = pVar u; fn = replace v (trm n)
      h <- (fn . q) <$> dig g [v]
      return ((n,h),u)
```

```
      rat = fmap (Tag Dig) stattr
      trm Trm {trName = "=", trArgs = [_,t]} = t; trm t = t
```

For technical reasons in the new parser `defNotion'` we have to expand the original parser `anotion`, defined by:

```
anotion = label "notion (at most one name)" $
  art >> gnotion basentn rat >>= single >>= hol
  where
    hol (q, f, v) = return (q, subst zHole (fst v) f)
    rat = fmap (Tag Dig) stattr
```

into

```
(q,f,w) <- (art >> gnotion basentn rat <|> fmap digadd primTvr) >>=
single
...
s <- fmap (Tag Dig) statement
      let f' = And f s
      let g = subst zHole (fst w) f'
...
```

The definition in Kelley can now be formalized like:

...

[synonym set/-s]

Definition 112. *x is a set iff for some y $x \in y$.*

14 ELEMENTARY ALGEBRA OF CLASSES

Definition 113. $x \cup y = \{set\ z\colon z \in x\ \ or\ z \in y\}$.

Definition 114. $x \cap y = \{set\ u\colon u \in x\ \ and\ u \in y\}$.

...

Definition 115. *x is a subclass of y iff for each z if $z \in x$ then $z \in y$.*

Let $x \subset y$ stand for x is a subclass of y. Let x is contained in y stand for $x \subset y$.

...

...

14.1 Line-By-Line Comments on `defNotion'`

We follow a successful parse of the following definition in Kelley:

Definition 116. *x is a subclass of y iff for each z if $z \in x$ then $z \in y$.*

This introduces a new "dependent type", i.e., a dependent notion "subclass of ..." or internally `aSubclassOf(v0,y)` with the definition:

```
[Translation] (line 144 of "/home/koepke/Desktop/Kelley_Formalization/
Sections_1-56_13.ftl")
aClass(y)
[Translation] (line 144 of "/home/koepke/Desktop/Kelley_Formalization/
Sections_1-56_13.ftl")
forall v0 ((HeadTerm :: aSubclassOf(v0,y)) iff (aClass(v0) and forall v1
(aClass(v1) implies (aElementOf(v1,v0) implies aElementOf(v1,y)))))
```

The second formula is generated by:

```
defNotion' = do
  ((n,h),u) <- wellFormedCheck (ntnVars . fst) defn; uDecl <- makeDecl u
  return $ dAll uDecl $ Iff (Tag HeadTerm n) h
  where ...
```

So the defining equivalence is obtained by
- parsing the defining sentence with `defn` to get a tupel `((n,h),u)` consisting of the newly defined (dependent) notion symbol `n` (in this case `aSubclassOf(x,y)`), the defining formula `h` (`aClass(x) and forall z (aClass(z) implies (aElementOf(z,x) implies aElementOf(z,y)))`), and ...

TO BE CONTINUED

14.2 The Current Formalization

We save the new source as `src13` and the formalization as `Section_1-56_13.ftl`:

15 The Classification Axiom Scheme

Let $x, y, z, r, s, t, u, v, a, b, c, d, e$ stand for classes.
Let $a \neq b$ stand for a != b. Let $a \in b$ stand for a is an element of b.

Axiom 117. *For each x, y it is true that $x = y$ if and only if for each z $z \in x$ when and only when $z \in y$.*

[synonym set/-s]

Definition 118. *x is a set iff for some y $x \in y$.*

16 ELEMENTARY ALGEBRA OF CLASSES

Definition 119. $x \cup y = \{set\ z : z \in x\ or\ z \in y\}$.

Definition 120. $x \cap y = \{set\ u : u \in x\ and\ u \in y\}$.

Let the union of x and y stand for $x \cup y$. Let the intersection of x and y stand for $x \cap y$.

Theorem 121. $(z \in x \cup y\ iff\ z \in x\ or\ z \in y)\ and\ (z \in x \cap y\ iff\ z \in x\ and\ z \in y)$.

Theorem 122. $x \cup x = x\ and\ x \cap x = x$.

Theorem 123. $x \cup y = y \cup x$ *and* $x \cap y = y \cap x$.

Theorem 124. $(x \cup y) \cup z = x \cup (y \cup z)$ *and* $(x \cap y) \cap z = x \cap (y \cap z)$.

Theorem 125. $x \cap (y \cup z) = (x \cap y) \cup (x \cap z)$ *and* $x \cup (y \cap z) = (x \cup y) \cap (x \cup z)$.

Let $a \notin b$ stand for a is not an element of b.

Definition 126. $\sim x = \{set\ y\colon y \notin x\}$.

Let the complement of x stand for $\sim x$.

Theorem 127. $\sim(\sim x) = x$.

Theorem 128. $\sim(x \cup y) = (\sim x) \cap (\sim y)$ *and* $\sim(x \cap y) = (\sim x) \cup (\sim y)$.

Definition 129. $x \sim y = x \cap (\sim y)$.

Theorem 130. $x \cap (y \sim z) = (x \cap y) \sim z$.

Definition 131. $0 = \{set\ x\colon x \neq x\}$.

Let the void class stand for 0. Let zero stand for 0.

Theorem 132. $x \notin 0$.

Theorem 133. $0 \cup x = x$ *and* $0 \cap x = 0$.

Definition 134. $\mathcal{U} = \{set\ x\colon x = x\}$.

Let the universe stand for $\mathcal{U}$.

Theorem 135. $x \in \mathcal{U}$ *if and only if* x *is a set.*

Theorem 136. $x \cup \mathcal{U} = \mathcal{U}$ *and* $x \cap \mathcal{U} = x$.

Theorem 137. $\sim 0 = \mathcal{U}$ *and* $\sim \mathcal{U} = 0$.

Definition 138. $\bigcap x = \{set\ z\text{: }for\ each\ y\ if\ y \in x\ then\ z \in y\}$.

Definition 139. $\bigcup x = \{set\ z\text{: }for\ some\ y\ (z \in y\ and\ y \in x)\}$.

Let the intersection of x stand for $\bigcap x$. Let the union of x stand for $\bigcup x$.

Theorem 140. $\bigcap 0 = \mathcal{U}$ *and* $\bigcup 0 = 0$.

Definition 141. x *is a subclass of* y *iff for each* z *if* $z \in x$ *then* $z \in y$.

Let $x \subset y$ stand for x is a subclass of y. Let x is contained in y stand for $x \subset y$.

Lemma 142. $0 \subset 0$ *and* $0 \notin 0$.

Theorem 143. $0 \subset x$ *and* $x \subset \mathcal{U}$.

Theorem 144. $x = y$ *iff* $x \subset y$ *and* $y \subset x$.

Theorem 145. *If* $x \subset y$ *and* $y \subset z$ *then* $x \subset z$.

Theorem 146. $x \subset y$ *iff* $x \cup y = y$.

Theorem 147. $x \subset y$ *iff* $x \cap y = x$.

Theorem 148. *If* $x \subset y$ *then* $\bigcup x \subset \bigcup y$ *and* $\bigcap y \subset \bigcap x$.

Theorem 149. *If* $x \in y$ *then* $x \subset \bigcup y$ *and* $\bigcap y \subset x$.

17 EXISTENCE OF SETS

Axiom 150. *If* x *is a set then there is a set* y *such that for each* z *if* $z \subset x$ *then* $z \in y$.

Theorem 151. *If* x *is a set and* $z \subset x$ *then* z *is a set.*

Theorem 152. $0 = \bigcap \mathcal{U}$ *and* $\mathcal{U} = \bigcup \mathcal{U}$.

Theorem 153. *If* $x \neq 0$ *then* $\bigcap x$ *is a set.*

Definition 154. $2^x = \{set\ y\colon y \subset x\}$.

Theorem 155. $\mathcal{U} = 2^{\mathcal{U}}$.

Theorem 156. *If* x *is a set then* 2^x *is a set and for each* y $y \subset x$ *iff* $y \in 2^x$.

Proof. Let x be a set. Take a set y such that for each z if $z \subset x$ then $z \in y$. $2^x \subset y$. □

Definition 157. $\mathcal{R} = \{set\ x\colon x \notin x\}$.

Theorem 158. $\mathcal{R}$ *is not a set.*

Theorem 159. $\mathcal{U}$ *is not a set.*

Definition 160. $\{x\} = \{set\ z\colon if\ x \in \mathcal{U}\ then\ z = x\}$.

Let the singleton of x stand for x.

Theorem 161. *If* x *is a set then for each* y $y \in \{x\}$ *iff* $y = x$.

Theorem 162. *If* x *is a set then* $\{x\}$ *is a set.*

Proof. Let x be a set. Then $\{x\} \subset 2^x$ and 2^x is a set. □

Theorem 163. $\{x\} = \mathcal{U}$ *if and only if* x *is not a set.*

Theorem 164. *If* x *is a set then* $\bigcap \{x\} = x$ *and* $\bigcup \{x\} = x$.

Theorem 165. *If x is not a set then $\bigcap\{x\}=0$ and $\bigcup\{x\}=\mathcal{U}$.*

Axiom 166. *If x is a set and y is a set then $x\cup y$ is a set.*

Definition 167. $\{x,y\}=\{x\}\cup\{y\}$.

Let the unordered pair of x and y stand for $\{x,y\}$.

Theorem 168. *If x is a set and y is a set then $\{x,y\}$ is a set and ($z\in\{x,y\}$ iff $z=x$ or $z=y$).*

Theorem 169. *$\{x,y\}=\mathcal{U}$ if and only if x is not a set or y is not a set.*

Theorem 170. *If x,y are sets then $\bigcap\{x,y\}=x\cap y$ and $\bigcup\{x,y\}=x\cup y$.*

Proof. Let x,y be sets. $\bigcup\{x,y\}\subset x\cup y$. $x\cup y\subset\bigcup\{x,y\}$. □

Theorem 171. *If x is not a set or y is not a set then $\bigcap\{x,y\}=0$ and $\bigcup\{x,y\}=\mathcal{U}$.*

18 ORDERED PAIRS: RELATIONS

Definition 172. $[x,y]=\{\{x\},\{x,y\}\}$.

Let the ordered pair of x and y stand for $[x,y]$.

Theorem 173. *$[x,y]$ is a set if and only if x is a set and y is a set.*

Theorem 174. *If $[x,y]$ is not a set then $[x,y]=\mathcal{U}$.*

Theorem 175. *If x and y are sets then $\bigcup[x,y]=\{x,y\}$ and $\bigcap[x,y]=\{x\}$ and $\bigcup\bigcap[x,y]=x$ and $\bigcap\bigcap[x,y]=x$ and $\bigcup\bigcup[x,y]=x\cup y$ and $\bigcap\bigcup[x,y]=x\cap y$.*

Theorem 176. *If x is not a set or y is not a set then $\bigcup\bigcap[x,y]=0$ and $\bigcap\bigcap[x,y]=\mathcal{U}$ and $\bigcup\bigcup[x,y]=\mathcal{U}$ and $\bigcap\bigcup[x,y]=0$.*

Definition 177. *1^{st} coord $z=\bigcap\bigcap z$.*

Definition 178. *2^{nd} coord $z=(\bigcap\bigcup z)\cup((\bigcup\bigcup z)\sim\bigcup\bigcap z)$.*

Let the first coordinate of z stand for 1^{st} coord z. Let the second coordinate of z stand for2^{nd} coord z.

Theorem 179. *2^{nd} coord $\mathcal{U}=\mathcal{U}$.*

Theorem 180. *If x and y are sets then 1^{st} coord $[x,y]=x$ and 2^{nd} coord $[x,y]=y$.*

Proof. Let x and y be sets. 2^{nd} coord $[x,y]=(\bigcap\bigcup[x,y])\cup((\bigcup\bigcup[x,y])\sim\bigcup\bigcap[x,y])=(x\cap y)\cup((x\cup y)\sim x)=y$. □

Theorem 181. *If x is not a set or y is not a set then 1^{st} coord $[x,y]=\mathcal{U}$ and 2^{nd} coord $[x,y]=\mathcal{U}$.*

Theorem 182. *If x and y are sets and $[x,y]=[u,v]$ then $x=u$ and $y=v$.*

Definition 183. *A class r is a relation if and only if for each element z of r there is x and y such that $z=[x,y]$.*