Formalization of the Fundamental Group in Untyped Set Theory Using Auto2

Bohua Zhan

Received: 26 February 2018 / Accepted: 25 July 2018 / Published online: 2 August 2018
© Springer Nature B.V. 2018

Abstract
We present a new framework for formalizing mathematics in untyped set theory using auto2. We demonstrate that many difficulties with using set theory for formalization of mathematics can be addressed by improvements to automation, without sacrificing the inherent flexibility of the logic. As applications, we formalize in Isabelle/FOL the entire chain of development from the axioms of set theory to the definition of the fundamental group of an arbitrary topological space. The auto2 prover is used as the sole automation tool, and enables succinct proof scripts throughout the project.

Keywords Isabelle · Set theory · Fundamental group

1 Introduction
Auto2 is a proof automation tool for the proof assistant Isabelle, introduced by the author in [19]. It uses a saturation-based proof search strategy at its core, and allows the user to easily add their own heuristics and proof procedures. With proper setup, it is intended to be able to consistently solve “routine” tasks encountered during a proof, thereby enabling a style of formalization using succinct proof scripts written in a purely declarative language.

In this paper, we present an application of auto2 to formalization of mathematics in untyped set theory1. In particular, we discuss several improvements to auto2 as well as strategies of usage that allow it to work effectively with untyped set theory. As examples, we formalize in Isabelle/FOL the entire chain of development from the axioms of set theory to the definition of the fundamental group for an arbitrary topological space, as well as prove several interesting results on the side.

The contribution of this paper is two-fold. First, we demonstrate that the auto2 system is capable of independently supporting proof developments on a relatively large scale. In [19], several case studies for auto2 were given in Isabelle/HOL. Each case study is at most several hundred lines long, and the use of auto2 is mixed with the use of other Isabelle tactics, as

1 Code available at https://github.com/bzhan/auto2

✉ Bohua Zhan
zhan@in.tum.de

1 Chair for Logic and Verification, Technical University of Munich, Munich, Germany
well as proof scripts provided by Sledgehammer. In contrast, the example we present in this paper is a unified development consisting of over 13,000 lines of theory files and 3,500 lines of ML code (not including the core auto2 program). The auto2 prover is used exclusively starting from basic set theory.

Second, we demonstrate that the additional complexity in proofs that arise when working with untyped set theory can be addressed through improvements to proof automation. For a number of reasons, untyped set theory is considered to be difficult to use for the purpose of formalized mathematics. For example, everything is represented as sets, including objects such as natural numbers that we usually do not think of as sets. Moreover, statements of theorems tend to be longer in untyped set theory than in typed theories, since assumptions that would otherwise be included in type constraints must now be stated explicitly. In this paper, we show that with appropriate definitions of basic concepts and setup for automation, all these complexities can be managed, without sacrificing the inherent flexibility of the logic.

We now give an outline for the rest of the paper. In Sect. 2, we describe how basic concepts in mathematics are defined using set theory in our work. In particular, we discuss the representation of functions and algebraic structures using extensible records, and the definition of quotient sets and operations on them. In Sect. 3, we describe our main ideas for automation in set theory. While these ideas are designed with auto2 in mind, they can potentially be applied in a wider class of proof automation systems. In Sect. 4, we review the basics of the auto2 system, and describe how the above ideas for automation in set theory can be implemented in this particular context.

In Secti. 5, we describe our main examples. We begin by giving two examples of proof scripts using auto2, taken from the proofs of the Schroeder-Bernstein theorem and a challenge problem in analysis from Lasse Rempe-Gillen. Next, we describe our main example: the definition of the fundamental group. Formalizing this definition requires reasoning about algebraic and topological structures, quotient sets, as well as continuous functions on real numbers. While this example has been formalized before in other systems, we believe it is a sufficiently challenging task with which to test the maturity of our framework. This section concludes with a discussion of performance and usability issues. Finally, we review related work in Sect. 6, and conclude in Sect. 7.

## 2 Basic Constructions in Set Theory

In this section, we discuss our choice of definitions of basic concepts, starting with the choice of logic. Our development is based on the FOL (first-order logic) instantiation of Isabelle. The initial parts are similar to those in Isabelle/ZF, and we refer to [15,16] for detailed explanations.

The only Isabelle types available are \( i \) for sets, \( o \) for propositions (booleans), and function types formed from them. We call objects with types other than \( i \) and \( o \) meta-functions, to distinguish them from functions defined within set theory (which have type \( i \)). It is possible to define higher-order meta-functions in FOL, and supply them with arguments in the form of lambda expressions. Theorems can be quantified over variables with functional type at the outermost level. These can be thought of as theorem-schemas in a first-order theory. However, one can only quantify over variables of type \( i \) inside the statement of a theorem, and the only equalities defined within FOL are those between types \( i \) (notation \( \cdot = \cdot \)) and \( o \) (notation \( \cdot \leftarrow \rightarrow \cdot \)). In practice, these restrictions mean that any functions that we wish to consider as first-class objects must be defined as set-theoretic functions.
2.1 Axioms of Set Theory

For uniformity of presentation, we start our development from FOL rather than theories in
Isabelle/ZF. However, the list of axioms that we use is mostly the same. The main addition is
the axiom of global choice, which we use as an easier-to-apply version of the axiom of choice.
Note that as in Isabelle/ZF, several of the axioms introduce new sets or meta-functions, and
declare properties satisfied by them. The exact list of axioms is as follows:

extension: \( \forall z. \; z \in x \iff z \in y \implies x = y \)
empty_set: \( x \notin \emptyset \)
collect: \( x \in \text{Collect}(A, P) \iff (x \in A \land P(x)) \)
upair: \( x \in \text{Upair}(y, z) \iff (x = y \lor x = z) \)
union: \( x \in \bigcup C \iff (\exists A \in C. \; x \in A) \)
power: \( x \in \text{Pow}(S) \iff x \subseteq S \)
replacement: \( \forall x \in A. \; \forall y. \; P(x, y) \land P(x, z) \implies y = z \implies \)
\( b \in \text{Replace}(A, P) \iff (\exists x \in A. \; P(x, b)) \)
foundation: \( \forall x. \; x \neq \emptyset \implies \exists y \in x. \; y \cap x = \emptyset \)
infinity: \( \emptyset \in \text{Inf} \land (\forall y \in \text{Inf}. \; \{y\} \cup y \in \text{Inf}) \)
choice: \( \exists x. \; x \in S \implies \text{Choice}(S) \in S \)

Next, we define several basic constructions in set theory. They are summarized in Table
1. See [15] for more explanations.

2.2 Functions as Records

The use of extensible records is one of the most important constructions in our framework. It
is used to represent both structures such as groups and topological spaces, as well as mappings
between sets and structures. It is often advantageous for different kinds of records to share
certain fields. For example, groups and rings should share the multiplication operator, rings
and ordered rings should share both addition and multiplication operators, and so on.

In set theory, two main approaches to representing records are tuples and partial functions.
We take the latter approach because we believe it carries more benefits in the long run. In this
approach, each field is identified with a natural number, so that two fields that can potentially
appear in the same structure are assigned different natural numbers. Structures are represented
as partial functions (given by sets of pairs) from fields to the corresponding data. As natural
numbers has not been formally defined at the time where we introduce structures, we use the
more explicit form \( \emptyset \), \( \text{succ}(\emptyset) \) to denote 0, 1, etc.

| Table 1 | Basic constructions in set theory |
|---------|----------------------------------|
| Notation | Definition                        |
| THE x. P(x) | \( \bigcup (\text{Replace}((\emptyset), \lambda x. \; P(y))) \) |
| \( \langle b(x) \rangle \; \; x \in A \) | \( \text{Replace}(A, \lambda x. \; y = b(x)) \) |
| SOME x \in A. P(x) | \( \text{Choice}((x \in A. \; P(x))) \) |
| \{a,b\} | \( \{a\}, \{a,b\} \) |
| \( \text{fst}(p) \) | \( \text{THE} \; a. \; \exists b. \; p = \langle a,b \rangle \) |
| \( \text{snd}(p) \) | \( \text{THE} \; b. \; \exists a. \; p = \langle a,b \rangle \) |
| \( \langle a_1, \ldots, a_n \rangle \) | \( \langle a_1, \langle a_2, \cdots, a_n \rangle \rangle \) |
| if \( P \) then \( a \) else \( b \) | \( \text{THE} \; z. \; P \land z=a \lor \neg P \land z=b \) |
| \( \bigcup a \in I. \; x \) | \( \bigcup (X(a). \; a \in I) \) |
| \( A \times B \) | \( x \in A. \; \bigcup y \in B. \; (x,y) \) |
We begin with the example of functions. A function is a record consisting of a source set (domain), a target set (codomain), and the graph of the function. In particular, we consider two functions with the same graph but different target sets to be different functions (another structure called family is used to represent functions without specified target set). The three fields are assigned to positions 0, 1, and 2 respectively:

\[
\begin{align*}
\text{definition} & \quad \text{"source_name} = \emptyset \text{"} \\
\text{definition} & \quad \text{"target_name} = \text{succ}(\emptyset) \text{"} \\
\text{definition} & \quad \text{"graph_name} = \text{succ(succ(\emptyset))} \text{"}
\end{align*}
\]

The accessors are defined using graph evaluation:

\[
\begin{align*}
\text{definition} & \quad \text{"source}(F) = \text{graph_eval}(F, \text{source_name}) \text{"} \\
\text{definition} & \quad \text{"target}(F) = \text{graph_eval}(F, \text{target_name}) \text{"} \\
\text{definition} & \quad \text{"graph}(F) = \text{graph_eval}(F, \text{graph_name}) \text{"}
\end{align*}
\]

For a graph \(G\) to represent a function from source \(S\) to target \(T\), it must satisfy the conditions for a functional graph:

\[
\begin{align*}
\text{definition} & \quad \text{is_func_graph} :: \quad "i \Rightarrow i \Rightarrow o" \text{ where } \text{is_func_graph}(G,X) \leftarrow \rightarrow \\
\text{definition} & \quad \text{func_graphs} :: \quad "i \Rightarrow i \Rightarrow i" \text{ where } \\
\text{definition} & \quad \text{function_space} :: \quad "i \Rightarrow i \Rightarrow i" \text{ (infixr "\rightarrow" 60) where } \\
\text{definition} & \quad \text{feval} :: \quad "i \Rightarrow i \Rightarrow i" \text{ (infixl "\^\_\" 90) where }
\end{align*}
\]

\[
\begin{align*}
\text{definition} & \quad \text{Fun} :: \quad "[i, i, i \Rightarrow i] \Rightarrow i" \text{ where } \\
\text{definition} & \quad \text{G ∈ func_graphs(A,B)}
\end{align*}
\]

The set of all functions from \(A\) to \(B\) (denoted \(A \rightarrow B\)) is then given by:

\[
\begin{align*}
\text{definition} & \quad \text{function_space} :: \quad "i \Rightarrow i \Rightarrow i" \text{ (infixr "\rightarrow" 60) where } \\
\text{definition} & \quad \text{feval} :: \quad "i \Rightarrow i \Rightarrow i" \text{ (infixl "\^\_\" 90) where }
\end{align*}
\]

\[
\begin{align*}
\text{definition} & \quad \text{Fun} :: \quad "[i, i, i \Rightarrow i] \Rightarrow i" \text{ where } \\
\text{definition} & \quad \text{G ∈ func_graphs(A,B)}
\end{align*}
\]

2.3 Algebraic Structures

A second major use of records is to represent algebraic structures. In our work, we will need to define structures such as groups, abelian groups, rings, and ordered rings. They share some common operations. For example, all these structures have an underlying set called carrier set. Several of them have the addition operator and/or the multiplication operator, and so on. Each of these operations is assigned to a distinct natural number. The assignments are given as follows (which shows there is no need for the assignments to be consecutive).

\[
\begin{align*}
\text{definition} & \quad \text{"carrier_name} = \emptyset \text{"} \\
\text{definition} & \quad \text{"order_graph_name} = \text{succ(succ(\emptyset))} \text{"} \\
\text{definition} & \quad \text{"zero_name} = \text{succ(succ(succ(\emptyset)))} \text{"} \\
\text{definition} & \quad \text{"plus_fun_name} = \text{succ(succ(succ(succ(\emptyset))))} \text{"} \\
\text{definition} & \quad \text{"one_name} = \text{succ(succ(succ(succ(succ(\emptyset))))}) \text{"} \\
\text{definition} & \quad \text{"times_fun_name} = \text{succ(succ(succ(succ(succ(succ(\emptyset)))))))} \text{"}
\end{align*}
\]
The accessors carrier, order_graph, zero, plus_fun, one, and times_fun are defined using graph evaluation on the corresponding names, as in the previous section.

Here order_graph is assumed to be a subset of \( S \times S \), and plus_fun, times_fun are assumed to be elements of \( S \times S \to S \). Hence, the operators \( \leq, +, \) and \( \ast \) can be defined as follows:

\[
\begin{align*}
\text{definition} & \quad \text{"le}(R,x,y) \iff (x,y)\in\text{order_graph}(R) \\
\text{definition} & \quad \text{"plus}(R,x,y) = \text{plus_fun}(R)(x,y)
\end{align*}
\]

These are abbreviated to \( x \leq R y \), \( x + R y \), and \( x \ast R y \), respectively (in both theory files and throughout this paper, we use \( \ast \) to denote multiplication in groups and rings, and \( \times \) to denote product on sets and other structures). We also abbreviate \( x \in \text{carrier}(S) \) to \( x \in S \).

The constructor for group-like structures is as follows:

\[
\begin{align*}
\text{definition} & \quad \text{Group} :: \[i, i, i \Rightarrow i \Rightarrow i \] \Rightarrow i \\
& \quad \text{where} \\
& \quad \text{Group}(S,u,f) = \text{Struct}({}\langle \text{carrier}_\text{name},S \rangle, \langle \text{one}_\text{name},u \rangle, \langle \text{times}_\text{fun}_\text{name}, \text{Fun}(S \times S, S, \lambda p. f(\text{fst}(p),\text{snd}(p))) \rangle})
\end{align*}
\]

The following predicate asserts that a structure contains at least the fields of a group-like structure, with the right membership properties (\( 1_g \) abbreviates \( \text{one}(G) \)):

\[
\begin{align*}
\text{definition} & \quad \text{is_group}_\text{raw} :: i \Rightarrow o \\
& \quad \text{where} \\
& \quad \text{is_group}_\text{raw}(G) \iff 1_g \in G \land \text{times}_\text{fun}(G) \in \text{carrier}(G) \times \text{carrier}(G) \to \text{carrier}(G)
\end{align*}
\]

To check whether such a structure is in fact a monoid / group, we use the following predicates:

\[
\begin{align*}
\text{definition} & \quad \text{is_monoid} :: i \Rightarrow o \\
& \quad \text{where} \\
& \quad \text{is_monoid}(G) \iff \text{is_group}_\text{raw}(G) \land \\
& \quad \quad (\forall x \in G. \forall y \in G. \forall z \in G. (x \ast_g y) \ast_g z = x \ast_g (y \ast_g z)) \land \\
& \quad \quad (\forall x \in G. 1_g \ast_g x = x \land x \ast_g 1_g = x)
\end{align*}
\]

\[
\begin{align*}
\text{definition} & \quad \text{units} :: i \Rightarrow i \\
& \quad \text{where} \\
& \quad \text{units}(G) = \{x \in G. (\exists y \in G. y \ast_g x = 1_g \land x \ast_g y = 1_g)\}
\end{align*}
\]

\[
\begin{align*}
\text{definition} & \quad \text{is_group} :: i \Rightarrow o \\
& \quad \text{where} \\
& \quad \text{is_group}(G) \iff \text{is_monoid}(G) \land \text{carrier}(G) = \text{units}(G)
\end{align*}
\]

Note these definitions are meaningful on any structure that has multiplicative data. Likewise, we can define a predicate \( \text{is_abgroup} \) for abelian groups, that is meaningful for any structure that has additive data. These can be combined with distributive properties to define the predicate for a ring:

\[
\begin{align*}
\text{definition} & \quad \text{is_ring} :: i \Rightarrow o \\
& \quad \text{where} \\
& \quad \text{is_ring}(R) \iff \text{is_ring}_\text{raw}(R) \land \text{is_abgroup}(R) \land \text{is_monoid}(R) \land \\
& \quad \quad \text{is_left_distrib}(R) \land \text{is_right_distrib}(R) \land 0_R \neq 1_R
\end{align*}
\]

Likewise, we can define the predicate for ordered rings, and constructors for such structures. Structures are used to represent the hierarchy of numbers: we let \( \text{nat}, \text{int}, \text{rat}, \) and \( \text{real} \) denote the set of natural numbers, integers, etc, while \( \mathbb{N}, \mathbb{Z}, \mathbb{Q}, \) and \( \mathbb{R} \) denote algebraic structures whose carrier sets are \( \text{nat}, \text{int}, \text{rat}, \) and \( \text{real} \), respectively. Hence, addition on natural numbers is denoted by \( x + \text{N} y \), addition on real numbers by \( x + \text{R} y \), etc. We can also state and prove theorems such as \( \text{is_ord_field}(\mathbb{R}) \), which contains all proof obligations for showing that the real numbers form an ordered field.
2.4 Morphism Between Structures

As we get into more advanced mathematics, we will frequently think of functions not just as a mapping between sets, but as a mapping between structures. For example, the concept for a mapping to be a group homomorphism is only meaningful when we fix group structures on its domain and codomain. We define morphisms as an extension of functions, with additional fields specifying structures on the source and target sets. The two additional fields are assigned to natural numbers 3 and 4 respectively:

- definition "source_str_name = succ(succ(succ(∅)))"
- definition "target_str_name = succ(succ(succ(succ(∅))))"
- definition "source_str(F) = graph_eval(F,source_str_name)"
- definition "target_str(F) = graph_eval(F,target_str_name)"

The constructor for a morphism is defined as follows (here \(S\) and \(T\) are the source and target structures, while the source and target sets are automatically derived):

- definition Mor :: "[i, i, i ⇒ i] ⇒ i" where
  "Mor(S,T,b) = (let A = carrier(S) in let B = carrier(T) in
  Struct({⟨source_name,A⟩,⟨target_name,B⟩,
  ⟨graph_name,\{p∈A×B. snd(p) = b(fst(p))\}⟩,
  ⟨source_str_name,S⟩,⟨target_str_name,T⟩})")"

The space of morphisms (denoted \(S \rightarrow T\)) is given by:

- definition mor_space :: "i ⇒ i ⇒ i" (infix "⇀" 60) where
  "mor_space(S,T) = (let A = carrier(S) in let B = carrier(T) in
  {Struct({⟨source_name,A⟩,⟨target_name,B⟩,⟨graph_name,G⟩,
  ⟨source_str_name,S⟩,⟨target_str_name,T⟩}).G ∈ func_graphs(A,B)})"

Note the notation \(f '/terms x\) for evaluation still works for morphisms. Several other concepts defined in terms of evaluation, such as image and inverse image, continue to be valid for morphisms as well, as are lemmas about these concepts. However, operations that construct new morphisms, such as inverse and composition, must be redefined. We will use \(g \circ f\) to denote the composition of two functions, and \(g \circ_m f\) to denote the composition of two morphisms.

Having morphisms store the source and target structures means we can define properties such as homomorphism on groups as a predicate:

- definition is_group_hom :: "i ⇒ o" where
  "is_group_hom(f) ≡ (let S = source_str(f) in let T = target_str(f) in
  is_morphism(f) ∧ is_group(S) ∧ is_group(T) ∧
  (∀x∈S. ∀y∈S. f(S x S y) = T f x T f y))"

The fact that the composition of two homomorphisms is a homomorphism can then be stated in a simple manner:

- lemma group_hom-compose: "is_group_hom(f) ⇒ is_group_hom(g) ⇒
  target_str(f) = source_str(g) ⇒ is_group_hom(g \circ_m f)"

2.5 Quotients and Equivalence Classes

The definition of quotients is another important task when building a framework for formalization of mathematics. In our work, we choose the following simple but effective approach.
First, equivalence relation is defined as a non-extensible record. Given an equivalence relation \( R \) on its carrier set \( S \), the quotient \( S/R \) is defined as the collection of equivalence classes of \( R \). Usually we want to define the quotient not just as a set but as a structure. This is achieved by defining a structure whose carrier set is \( S/R \), and whose operations are functions on \( S/R \).

In more detail, the following definitions set up a non-extensible record for equivalence relations:

\[
\text{definition } \equiv_{\text{graph\_name}} = \text{succ}(\text{succ}(\emptyset))
\]
\[
\text{definition } \equiv_{\text{graph}}(R) = \text{graph\_eval}(R, \equiv_{\text{graph\_name}})
\]
\[
\text{definition } \text{raw\_equiv}(R) \iff \text{is\_func\_graph}(R, \{\text{carrier\_name}, \equiv_{\text{graph\_name}}\}) \\
\quad \land \equiv_{\text{graph}}(R) \in \text{Pow}(\text{carrier}(R) \times \text{carrier}(R))
\]
\[
\text{definition } \text{Equiv}(S,R) = \text{Struct}(\{\langle \text{carrier\_name}, S \rangle, \langle \equiv_{\text{graph\_name}}, \text{rel\_graph}(S,R) \rangle\})
\]
\[
\text{definition } x \sim_R y \iff \langle x, y \rangle \in \equiv_{\text{graph}}(R)
\]

The \( \equiv \) predicate asserts the usual properties of an equivalence relation:

\[
\text{definition } \equiv : \{i \Rightarrow o\} \text{ where }
\]
\[
\equiv(R) \iff \text{raw\_equiv}(R) \land (\forall x \in R. x \sim_R x) \\
\quad \land (\forall x y z. x \sim_R y \rightarrow y \sim_R z \rightarrow x \sim_R z)
\]

Given an equivalence relation \( R \), we define the equivalence class of an element and then the quotient set:

\[
\text{definition } \equiv_{\text{class}}(R, x) = \{y \in R. x \sim_R y\}
\]
\[
\text{definition } E / R = \{\equiv_{\text{class}}(R, x). x \in E\}
\]

In the latter definition, we assume that \( E \) is always the carrier set of \( R \). Both are written for the sake of clarity. The meta-function \( \equiv_{\text{class}} \) provides the way to obtain, from any element in \( E \), its corresponding element in \( E/R \). The following defines a set-theoretic function for this purpose:

\[
\text{definition } q\text{surj} : \{i \Rightarrow i\} \text{ where }
\]
\[
q\text{surj}(R) = \text{Fun}(\text{carrier}(R),\text{carrier}(R)/R, \lambda x. \equiv_{\text{class}}(R, x))
\]

Finally, we need a way to obtain, for any element in \( \text{carrier}(R)/R \), a representative of that element in \( \text{carrier}(R) \). The choice function is ideal for this purpose:

\[
\text{definition } \text{rep}(R, x) = \text{Choice}(x)
\]

Using this definition does have the disadvantage that the axiom of choice is used in places where it may not be necessary. However, for us the convenience it provides outweights these concerns.

After defining the quotient set, we can define operations on it using \( \equiv_{\text{class}} \) and \( \text{rep} \). The fact that the definition does not depend on the choice of the representative can be proved afterwards as a theorem.

As a concrete example, consider the definition of the quotient group. Given a group \( G \) and a subset \( H \) of \( G \) (later assumed to be a normal subgroup), we first define the equivalence relation for cosets:

\[
\text{definition } r\text{coset\_equiv} : \{i \Rightarrow i\} \text{ where }
\]
\[
r\text{coset\_equiv}(G,H) = \text{Equiv}(\text{carrier}(G), \lambda x y. \exists h\in H. h*g x = y)
\]

This is an equivalence relation whenever \( H \) is a subgroup:

\[
\text{lemma } r\text{coset\_equiv\_is\_equiv}:
\]
\[
\text{is\_group}(G) \rightarrow \text{is\_subgroup\_set}(G,H) \rightarrow \\
r\text{coset\_equiv}(G,H) \in \text{equiv\_space}(\text{carrier}(G))
\]
The quotient set (which will be made into a group later) is defined as:

**Definition** `rcoset_quot :: "i ⇒ i ⇒ i" where``

`rcoset_quot(G,H) = carrier(rcoset_equiv(G,H)) // rcoset_equiv(G,H)`

Multiplication on the quotient set is defined as follows:

**Definition** `rcoset_mult :: "i ⇒ i ⇒ i ⇒ i ⇒ i" where``

`rcoset_mult(G,H,x,y) = (let R = rcoset_equiv(G,H) in
equiv_class(R, rep(R,x) *G rep(R,y)))`

Here, `x` and `y` are assumed to be elements in `rcoset_quot(G,H)`. In general (e.g. when `H` is a subgroup but not a normal subgroup of `G`), the above definition depends on which representative is chosen in `rep(R,x)` and `rep(R,y)`. Hence, the definition is mathematically ambiguous, but perfectly valid under our choice of foundations. To go further, however, we need to show that under the right conditions, the definition does not depend on the choice made in `rep(R,x)` and `rep(R,y)`. This can be expressed in the following lemma:

**Lemma** `rcoset_mult_eval`:

`R = rcoset_equiv(G,H) ⇒ x ∈ R ⇒ y ∈ R ⇒ is_normal_subgroup_set(G,H) ⇒ rcoset_mult(G,H,equiv_class(R,x),equiv_class(R,y)) = equiv_class(R,x * y)`

Note the additional assumption that `H` is a normal subgroup of `G`. With this lemma, we can begin to prove the usual properties of `rcoset_mult` (e.g. associativity).

The identity element in the quotient group is simply the equivalence class of the identity in the original group:

**Definition** `rcoset_id :: "i ⇒ i ⇒ i" where``

`rcoset_id(G,H) = equiv_class(rcoset_equiv(G,H), 1_G)`

Putting everything together, the definition of the quotient group is as follows:

**Definition** `quotient_group :: "i ⇒ i ⇒ i" (infix "/\ G" 90) where``

`G //\ G H = Group(rcoset_quot(G,H), rcoset_id(G,H), λx y. rcoset_mult(G,H,x,y))`

Finally, the fact that the quotient group is a group is stated as follows (here the condition `is_normal_subgroup_set(G,H)` contains the assumption that `G` is a group):

**Lemma** `quotient_group_is_group`:

`is_normal_subgroup_set(G,H) ⇒ is_group(G //\ G H)`

This example also demonstrates that the construction of equivalence classes and quotient sets can be parametrized over structures and other inputs. It serves as a model for the (more complicated) construction of the fundamental group later.

### 3 Automation in Untyped Set Theory

In this section, we describe how some of the difficulties associated with using untyped set theory for formalization of mathematics can be addressed by improvements to automation. This can be roughly divided into two parts: abstraction from internal representations (Sect. 3.1) and management of side conditions (Sects. 3.2 to 3.4).

While these strategies are designed with auto2 in mind, they can potentially be implemented in other systems of proof automation. The only ingredients that appear to be necessary are the use of saturation-based proof search and some mechanism for the user to annotate definitions and lemmas for the prover. Hence, this section is written to avoid details specific to auto2. Implementation details will be described in the next section.
3.1 Encapsulation of Definitions

One commonly cited problem with untyped set theory is that every object is a set, including those that are not usually considered as sets. Common examples of the latter include ordered pairs, natural numbers, functions, etc. In informal treatments of mathematics, the definitions of these concepts as sets are used only to establish their basic properties. Once these properties are proved, the definitions are never used again.

In formal developments, when automation is used to produce large parts of the proof, one potential problem is that the automation may needlessly expand the original definitions of objects, rather than focusing on their basic properties. This increases the search space and obscures the essential ideas of the proof. In our framework, this problem is addressed using the ability to remove reasoning rules for the prover. For any definition whose only purpose is to construct the object with the right properties, we use the following three-step procedure:

1. The definition is stated and added to the prover as rewrite rules.
2. Basic properties of the object being defined are stated and proved. These properties are added to the prover as appropriate reasoning rules.
3. The rewrite rules for the original definition are removed from use by the prover.

We emphasize that the original definitions are only removed from the proof automation. They are still true statements in the logical sense, and can be added back to the automation at any time, or directly invoked when implementing specific proof procedures.

We demonstrate this method with two examples: ordered pairs and structures.

3.1.1 Ordered Pairs

The definition of ordered pairs in set theory is famous for seeming arbitrary, the only purpose being to achieve the right properties. We follow the standard definition in our work:

definition "\langle a,b \rangle = \{ \{a\}, \{a,b\}\}"

definition "fst(p) = (THE a. \exists b. p = \langle a,b \rangle)"

definition "snd(p) = (THE b. \exists a. p = \langle a,b \rangle)"

With these definitions added to the proof automation, it should be able to help in proving the basic properties of ordered pairs:

lemma pair_eqD: "\langle a,b \rangle = \langle c,d \rangle \Longrightarrow a = b \land c = d"

lemma fst_conv: "fst(\langle a,b \rangle) = a"

lemma snd_conv: "snd(\langle a,b \rangle) = b"

Once these lemmas are added to the automation (for example, the first lemma as a forward reasoning rule, and the latter two lemmas as rewrite rules), the original definitions of \langle·,·\rangle, fst, and snd can be removed. In this way, the automation can be used in any proof that depends only on the basic properties of ordered pairs. It will lose the ability to discover proofs that depend on the original definitions. In particular, it will no longer be able to make sense of facts of the form \(x \in \langle a,b \rangle\), or rewrite the term \(fst(p)\) unless \(p\) is known to be an ordered pair. However, in some sense this is exactly the intended behavior, as the latter statements would be considered “ill-typed” in the usual development of mathematics.

3.1.2 Structures

A more advanced example is the encapsulation of structures. As described in Sect. 2, structures are represented as partial functions from natural numbers. Our goal is to encapsulate
this definition, so that the representation as partial functions would not be expanded in the usual reasoning with structures. However, when new forms of structures are defined, we would like to automatically prove the basic properties, e.g. relations between constructors and accessors, and so on.

As an example, consider the constructor \texttt{Group} for group-like structures. We repeat the definition here:

\begin{verbatim}
definition Group :: "[i, i, i ⇒ i ⇒ i] ⇒ i" where
"Group(S,u,f) = Struct({⟨carrier_name,S⟩,⟨one_name,u⟩,⟨times_fun_name, Fun(S×S, S, λp. f(fst(p),snd(p)))⟩})"
\end{verbatim}

Given this definition, we would like to prove the following:

\begin{verbatim}
lemma group_eval:
"carrier(Group(S,u,f)) = S"
"one(Group(S,u,f)) = u"
"G = Group(S,u,f) /⇒ x ∈ .G /⇒ y ∈ .G /⇒ is_group_raw(G) /⇒ x *G y = f(x,y)"
\end{verbatim}

After expanding the definition of \texttt{Group}, the proof of the above equalities involves expanding the definitions of \texttt{carrier}, \texttt{carrier_name}, etc, and makes use of the fact that the three field names are assigned to distinct natural numbers. This proof takes many steps but follow a fixed procedure. Hence, it can be implemented as a proof procedure to be invoked upon seeing the marker \texttt{Struct}.

Another basic property of structures is that given two structures with the same fields, if the data at all fields are equal, then the two structures are equal. The proof of this also follows a fixed procedure and can be implemented as such. The details are omitted here.

We notice that the definitions of accessors like \texttt{carrier}, and of field names like \texttt{carrier_name}, never need to be used directly by the automation, but only as a part of some proof procedure. Hence, they never need to be added as rewrite rules. After defining a constructor such as \texttt{Group}, the basic properties of \texttt{Group} can be proved using automatic procedures. After that, the definition of \texttt{Group} can be hidden as well, so its representation as partial functions will not be expanded in any future reasoning.

### 3.2 Property Table

In this and the next section, we discuss two tables of derived facts that are maintained by the prover during a proof. The main motivation for the two tables is that for many theorems, especially those stated in an untyped logic, some of its assumptions can be considered as “side conditions”. As an example, consider the following lemma:

\begin{verbatim}
lemma unit_l_cancel:
"is_monoid(G) /⇒ y ∈ .G /⇒ z ∈ .G /⇒ x ∈ units(G) /⇒ x *G y = x *G z /⇒ x ∈ units(G) /⇒ y = z"
\end{verbatim}

In this lemma, the last two assumptions are the “\textit{main}” assumptions, while the first three are side conditions asserting that the variables in the main assumptions are well-behaved in some sense. In Isabelle/HOL, these side conditions may be folded into type or type-class constraints.

We consider two kinds of side conditions. The first kind, like the first assumption in the lemma above, checks that one of the variables in the main assumptions satisfies a certain predicate. In Isabelle/HOL, these may correspond to type-class constraints. In our framework, we call these \textit{property assumptions}. More precisely, we call any constant unary predicate (in
FOL this means a constant of type $i \Rightarrow o$ a property. The property table records, during a proof, the list of properties satisfied by each term currently appearing in the proof. Properties propagate through equalities: if $P(a)$ is in the property table, and the equality $a = b$ is known (with $b$ also appearing in the proof), then $P(b)$ should be added to the property table.

There are two types of theorems that the user can add as additional propagation rules for the property table. The first type derives a property for a term $t$ from other properties of $t$. As a simple example, the statement that any group is a monoid:

```
lemma is_groupD: "is_group(G) $\Rightarrow$ is_monoid(G)".
```

The theorem can have multiple assumptions. For example, the following theorem, expressing a major result in real analysis, is also a property propagation rule:

```
lemma complete_to_linear_continuum:
  "cauchy_complete_field(R) $\Rightarrow$ is_archimedean(R) $\Rightarrow$ linear_continuum(R)".
```

These propagation rules are intended to emulate the lattice of inclusions for type classes in Isabelle/HOL. The second type of property propagation rules derive a property for a term $f(x_1, \ldots, x_n)$ appearing in the proof from properties of the arguments $x_1, \ldots, x_n$. A simple example is the fact that the product of two groups is a group:

```
lemma group_prod_is_group:
  "is_group(G) $\Rightarrow$ is_group(H) $\Rightarrow$ is_group(G $\times$ H)"
```

These propagation rules are intended to emulate type-class inference rules for type constructors in Isabelle/HOL. The propagation of properties in these two cases are also similar to the processing of Mizar adjective cluster registrations. In our framework, the proof automation can access the current list of property propagation rules, and is responsible for applying them automatically during a proof.

### 3.3 Well-Formedness Table

The second kind of side conditions assert that certain terms occurring in the main assumptions of a theorem are well-formed. We use the terminology of well-formedness to capture a familiar feature of mathematical language: that an expression may make implicit assumptions about its subterms. These conditions can be in the form of type constraints. For example, the expression $a + R b$ implicitly assumes that $a$ and $b$ are elements in the carrier set of $R$. However, this concept is much more general. Some examples of well-formedness conditions are summarized in Table 2.

In general, given any meta-function $f$, any propositional expression in terms of the arguments of $f$ can be registered as a well-formedness condition of $f$. In particular, well-formedness conditions are not necessarily properties. For example, the well-formedness condition $a \in R$ for the term $a + R b$ involves two variables and hence is not a property. The well-formedness table records, for every term encountered so far in the proof, the list of its well-formedness conditions that are satisfied. Whenever a new fact is derived, the automation should check whether it verifies a well-formedness condition of an existing term.

The property and well-formedness tables allow the proof automation to quickly find the required side conditions for applying a lemma, simplifying the process for applying lemmas. This is particularly important when working in untyped set theory, given the prevalence of side conditions corresponding to type and type-class constraints. From our experience, most of such constraints can indeed be expressed as side conditions. Hence, the use of property and well-formedness tables has the potential to eliminate the extra complexity of dealing with type assumptions when working in set theory.
Table 2  Examples of well-formedness conditions

| Term | Conditions |
|------|------------|
| \( \bigcap A \) | \( A \neq \emptyset \) |
| \( f \setminus x \) | \( x \in \text{source}(f) \) |
| \( g \circ f \) | \( \text{target}(f) = \text{source}(g) \) |
| \( g \circ_{\text{m}} f \) | \( \text{target}_{\text{str}}(f) = \text{source}_{\text{str}}(g) \) |
| \( a +_{R} b \) | \( a \in \mathbb{R}, \ b \in \mathbb{R} \) |
| \( \text{inv}(R, a) \) | \( a \in \text{units}(R) \) |
| \( a /_{R} b \) | \( a \in \mathbb{R}, \ b \in \text{units}(R) \) |
| \( \text{subgroup}(G,H) \) | \( \text{is_subgroup_set}(G,H) \) |
| \( \text{quotient_group}(G,H) \) | \( \text{is_normal_subgroup_set}(G,H) \) |

3.4 Well-Formed Conversions

Algebraic simplification is an important part of any system of proof automation. For many kinds of algebraic structures, e.g. monoids, groups, abelian groups, and rings, there is a concept of normal form of an expression, and equality between two terms can be decided by normalizing both sides. In untyped set theory, such computation of normal forms is complicated by the fact that the relevant rewriting rules have extra assumptions. For example, the rule for associativity of addition is:

\[
is_{\text{abgroup}}(R) \Rightarrow x \in \mathbb{R} \Rightarrow y \in \mathbb{R} \Rightarrow z \in \mathbb{R} \Rightarrow (x +_{R} (y +_{R} z)) = ((x +_{R} y) +_{R} z)
\]

The first assumption can be verified at the beginning of the normalization process. The remaining assumptions, however, are more cumbersome. In particular, they may require membership status of terms that arise only during the normalization. For example, when normalizing the term \( a +_{R} (b +_{R} (c +_{R} d)) \), we may first rewrite it to \( a +_{R} ((b +_{R} c) +_{R} d) \). The next step, however, requires \( b +_{R} c \in \mathbb{R} \), where \( b +_{R} c \) does not appear initially and may not have appeared so far in the proof. In typed theories, this poses no problem, since \( b +_{R} c \) will be automatically given the same type as \( b \) and \( c \) when the term is created.

In untyped set theory, such membership information must be kept track of and derived when necessary. The concept of well-formed terms provides a natural framework for doing this. Before performing algebraic normalization on a term, we first check for all relevant well-formedness conditions. If all conditions are present, we produce a data structure (of type \( \text{wfterm} \) in Isabelle/ML) containing the certified term as well as theorems asserting well-formedness conditions. A theorem is called a well-formed rewrite rule if the following holds:

- The first conclusion of the theorem is an equality.
- Each assumption of the theorem is a well-formedness condition for terms on the left side of the equality.
- The remaining conclusions of the theorem verify all well-formedness conditions for terms on the right side of the equality that are not already present in the assumptions.

For example, the associativity rule stated above is not yet a well-formed rewrite rule: there is no justification for \( x +_{R} y \in \mathbb{R} \), which is a well-formedness condition for the term \((x +_{R} y) +_{R} z\) on the right side of the equality. The full well-formed rewrite rule is:
Given a well-formed rewrite rule, we can produce a well-formed conversion that acts on $\text{wfterm}$ objects, in a way similar to how equalities produce regular conversions that act on $\text{c terme}$ objects in Isabelle/ML. Like regular conversions, well-formed conversions can be composed in various ways, and full normalization procedures can be written using the language of well-formed conversions. These normalization procedures in turn can be used to implement various proof procedures. We give two examples:

- Given two terms $s$ and $t$ that are non-atomic with respect to operations in $R$, where $R$ is a monoid (group / abelian group / ring), normalize $s$ and $t$ using the rules for $R$. If the normalizations are equal, output $s = t$.
- Given two propositions $a \leq_R b$ and $\neg (c \leq_R d)$, where $R$ is an ordered ring, compare the normalizations of $b -_R a$ and $d -_R c$. If they are equal, output a contradiction.

These proof procedures allow algebraic manipulations to be performed rapidly. They replace the matching of associative-commutative functions for HOL described in [19].

3.5 Discussion

We conclude this section with a discussion of our overall approach to untyped set theory, and compare it with other approaches. One feature of our approach is that we do not seek to re-institute a concept of types in our framework, but simply replace type constraints with set membership conditions (or predicates, for constraints that cannot be described by a set). The aim is to fully preserve the flexibility of set membership when compared to types. Empirically, most of the extra assumptions that arise in the statement of theorems can be taken care of by classifying them as properties or well-formedness conditions. Our approach can be contrasted with that taken by Mizar, which defines a concept of soft types [18] within the core of the system.

Every framework for formalizing modern mathematics need a way to deal with structures. In Mizar, structures are defined in the core of the system as partial functions on selectors [10,17]. In both Isabelle/HOL and IsarMathLib’s treatment of abstract algebra, structures are realized with extensive use of locales. For Coq, one notable approach is the use of Canonical Structures [11] in the formalization of the Odd Order Theorem. In the current version of our work, we follow Mizar’s approach, representing structures as partial functions from natural numbers.

Finally, we emphasize that we do not make any modification to Isabelle/FOL in our development. The concept of well-formed terms, for example, is meaningful only to the automation. While the proof automation may have concepts similar to types and type classes in its internal search, the resulting proof is still stated in the logic of untyped set theory. Hence, to have confidence in the proofs, one only need to trust the existing Isabelle system, the ten axioms stated in Sect. 2.1, and the definitions involved in the statement of the results.

4 Implementation in Auto2

In this section, we describe how the above ideas for automation in set theory is implemented in the auto2 prover, which is used exclusively as proof automation for this project. In the first two subsections, we review the basic ideas of auto2, mostly as presented in [19]. In the last
two subsections, we describe the modifications to auto2 that are necessary for implementing properties and well-formedness conditions.

4.1 Overview of Auto2

Auto2 is a theorem prover packaged as a tactic in Isabelle. It follows a saturation-based proof search. That is, proof progresses by successively adding to a list of facts that can be derived from the initial assumptions, until a contradiction is found. A collection of rules of reasoning called *proof steps* is used to derive new facts from old ones. New proof steps can be added and removed at any time within an Isabelle theory. Removing proof steps is infrequent in practice but can be used, for example, to realize the encapsulation of definitions in Sect. 3.1.

In addition to the list of derived facts, auto2 also maintains a *rewrite table* keeping track of currently derived equalities (without arbitrary variables). In particular, the table computes the congruence closure of the equalities. This is then used for E-matching (matching modulo to known equalities, see [13]) in many of the proof steps.

A priority queue is used to store the newly generated but unprocessed facts. Proof steps generate derived facts in the form of *updates*, which can be thought of as a package of facts to be processed together. Each update is assigned a score and added to the priority queue. Lower score means higher priority. In general, the score of an update is computed from the size of the facts, and the scores of facts from which it is derived.

We can now describe the overall algorithm of auto2. The data structures maintained during the proof are a list of processed facts, a priority queue of unprocessed facts, and a rewrite table maintaining the processed equalities. At every iteration, the prover removes the update with the highest priority from the priority queue. Any ground equalities in the update are added to the rewrite table, which is then completed to maintain the congruence closure. Next, all proof steps are applied, and any updates that result from the new fact or the new equalities are added to the priority queue. If there are no new equalities, only the new items need to be matched, possibly pairing with one of the existing items. Otherwise, all items that contain terms appearing on either sides of the new equalities must be matched again, as E-matching on these items may generate new matches.

One final complication comes from the need for case analysis. In addition to deriving new facts, proof steps can also add new case analysis. Case analysis is realized using a system of boxes. Each *primitive box* represents a case, recording the extra assumptions for that case. Each derived fact and entry in the rewrite table resides in the intersection of a list of primitive boxes (which is called a *box*), recording which extra assumptions the fact or equality is derived from. When a contradiction is derived in some box, it is marked as resolved, and appropriate facts are added to its parent boxes. The algorithm ends when a contradiction is derived in the box without extra assumptions.

The auto2 prover is not intended to be complete. For example, it may intentionally apply a theorem in only one of several possible directions, in order to reduce the search space. For more difficult theorems, auto2 provides a custom proof language, allowing the user to specify intermediate steps of the proof. Generally, when proving a result using auto2, the user will first try to prove it without providing any hints, and in case of failure, successively add intermediate steps, possibly following an informal proof of the result.

Currently, auto2 is set up to work with both Isabelle/HOL and Isabelle/FOL. It consists of a core program, and basic proof steps about predicate logic and equality, parametrized over the list of constants and theorems for the target logic.
4.2 Proof Steps

The list of proof steps represent the mathematical knowledge used by auto2. Each proof step is a function that takes one or two existing facts as input, and outputs updates that can add derived facts, create a case analysis, shadow a fact (due to redundancy), or resolve a box.

There are two broad categories of proof steps, which we call the standard and special proof steps in this paper. A standard proof step applies an existing theorem in a specific direction. It matches the input items to one or two patterns in the statement of the theorem, and applies the theorem to derive a new proposition. Here the matching is E-matching using the rewrite table. A special proof step can have more complex behavior, and is usually written as an ML function. In practice, the vast majority of proof steps are standard, although special proof steps also play an important role.

Standard proof steps can be added by setting an appropriate Isabelle attribute (attributes in Isabelle are annotations to a theorem that can be set when the theorem is stated, in a bracket after the name of the theorem, or at any later time in developing a theory). We describe the most frequently used attributes here. For each attribute, we specify which patterns will be matched. In all cases the proof step will apply the theorem in an appropriate manner for each match.

- **forward**: match up to two assumptions of the theorem, starting from the front.
- **backward**: match the conclusion.
- **backward1**: match the last assumption and the conclusion.
- **backward2**: match the first assumption and the conclusion.
- **resolve**: with the theorem having at most one assumption, match the possible assumption and the conclusion. Output a contradiction for each match.
- **rewrite**: with the conclusion of the theorem being an equality, match up to one assumption, starting from the front, and the left side of the equality.

4.3 Side Conditions in Standard Proof Steps

In this section, we describe how standard proof steps are modified in the presence of properties and well-formedness conditions.

Each of the basic attributes requires a certain minimum number of assumptions in the theorem for it to apply. For example, the attributes **forward** and **backward** require at least one assumption, the attributes **backward1** and **backward2** require at least two assumptions, and the attribute **rewrite** has no requirements (at least zero assumptions). When a lemma with \( n \) assumptions is added using an attribute that requires at least \( k \) assumptions, auto2 will, for each number \( i \) from 0 to \( n - k \), check whether it is possible to add the lemma with the first \( i \) assumptions as side conditions. This means figuring out the patterns to be matched under the attribute if the first \( i \) assumptions are not present, and check whether each of the first \( i \) assumptions can be considered as a property or well-formedness condition of some subterm of the match patterns. The largest value of \( i \) for which this is possible is then chosen. The added proof step will match the input items to the patterns. If there is a match, it will additionally lookup all side conditions in the property and well-formedness tables. Only if all side conditions are found, it will apply the theorem to output a new item.

As an example, consider the lemma **unit_l_cancel** given in Sect. 3.2:

```
lemma unit_l_cancel:
  "is_monoid(G) ⇒ y ∈ G ⇒ z ∈ G ⇒ x *_G y = x *_G z ⇒ x ∈ units(G) ⇒ y = z"
```
This lemma can be added as a forward rule. Auto2 will check, for each \( i = 0, 1, 2, 3, 4, \) whether the first \( i \) assumptions can be considered as side conditions. Indeed, it is possible to consider the first three assumptions as side conditions. In this case, the patterns \( x \ast_G y = x \ast_G z \) and \( x \in \text{units}(G) \) are matched. Then \( \text{is_monoid}(G) \) is a property of \( G \), and \( y \in \text{units}(G) \). \( G, z \in G \) are well-formedness conditions of \( x \ast_G y \) and \( x \ast_G z \), respectively.

Adding property propagation rules is also achieved by setting the \textit{forward} attribute. Given a lemma, it is possible to determine whether the lemma can be added as a property propagation rule. If \textit{forward} attribute is used, auto2 will first determine whether the lemma can be added as a property propagation rule. If yes, it is added as one, and no proof step is added. Otherwise, it proceeds to add the lemma as a regular forward proof step following the above procedure.

### 4.4 Side Conditions During Proof Search

The property table and the well-formedness table are maintained as additional tables by auto2 during the proof. This complicates the procedure at each iteration of the main loop. We now describe this in more detail.

Every iteration of the main loop adds one or more new items to the proof context. The ensuing procedure can be divided into two stages. In the first stage, the rewrite table, property table, and well-formedness table are updated to include all consequences of adding the new items. The rewrite table is updated first, adding any new equalities and completing the congruence closure as usual. The property table is updated next, so that property propagation will also be made along the newly introduced equalities. Finally, the well-formedness table is updated, by matching each of the unresolved well-formedness conditions with each of the new items, as well as each of the existing items that can potentially have new matches due to the new equalities.

In the second stage, all proof steps are applied to see if there are new updates as a result of new facts, or new entries to the rewrite, property, and well-formedness tables. If there are no new entries in the three tables, only the new items need to be matched, possibly pairing with one of the existing items. Otherwise, all items that contain terms appearing on either side of the new equalities, or has new properties or well-formedness conditions, need to be matched again. In the latter case, it is because side conditions that are not found in a previous application of some proof step may be found now, hence new updates could be available.

### 5 Examples

As application of auto2 and its setup for set theory, we formalized enough mathematics in Isabelle/FOL to be able to define the fundamental group. In addition to work directly used for that purpose, we also formalized several interesting results on the side. These include the well-ordering theorem and Zorn’s lemma, the first isomorphism theorem for groups, and the intermediate value theorem. Two more examples are presented in this section, to demonstrate the level of succinctness of proof scripts that can be achieved, and offer some comparisons with other systems. We emphasize, however, that the comparisons are only approximate, as a lot depend on the setup of auto2 in our work and on the style of proof in other systems.

Throughout our work, we referred to various sources including both mathematical texts and other formalizations. We list these sources here:
Axioms of set theory and basic operations on sets, construction of natural numbers using least fixed points: from Isabelle/ZF [15, 16].

Equivalence and order relations, arbitrary products on sets, well-ordering theorem and Zorn’s lemma: from Bourbaki’s Theory of Sets [3].

Group theory and the construction of real numbers using Cauchy sequences: from the author’s previous case studies [19], which in turn is based on corresponding theories in the Isabelle/HOL library.

Point-set topology and construction of the fundamental group: from Topology by Munkres [14].

5.1 Schroeder–Bernstein Theorem

For our first example, we present the proof of the Schroeder-Bernstein theorem. See [16] for a presentation of the same proof in Isabelle/ZF. The bijection is constructed by gluing together two functions. Auto2 is able to prove automatically that under certain conditions, the gluing is a bijection (lemma $\text{glue\_function2\_bij}$). For the Schroeder-Bernstein theorem itself, three intermediate steps (and a few definitions of variables) are needed.

\begin{verbatim}
\textbf{definition} \textit{glue\_function2} :: $\text{i} \Rightarrow \text{i} \Rightarrow \text{i}$ \textit{where}
"$\text{glue\_function2}(f, g) = \text{Fun} (\text{source}(f) \cup \text{source}(g), \text{target}(f) \cup \text{target}(g),"
\textit{\ lambda} x. \text{if} x \in \text{source}(f) \text{ then } f x \text{ else } g x)"
\end{verbatim}

\begin{verbatim}
\textbf{lemma} \textit{glue\_function2\_bij} [backward]:
"f \in A \cong B \Longrightarrow g \in C \cong D \Longrightarrow A \cap C = \emptyset \Longrightarrow B \cap D = \emptyset \Longrightarrow
\text{glue\_function2}(f, g) \in (A \cup C) \cong (B \cup D)* \text{ by auto2}
\end{verbatim}

\begin{verbatim}
\textbf{theorem} \textit{schroeder\_bernstein}:
"\text{injective}(f) \Longrightarrow \text{injective}(g) \Longrightarrow f \in X \rightarrow Y \Longrightarrow g \in Y \rightarrow X \Longrightarrow
\text{equipotent}(X, Y)"
\end{verbatim}

@proof
@let "X_A = lfp(X, \lambda W. X - g''(Y - f''W))"
@let "X_B = X - X_A, Y_A = f''X_A, Y_B = Y - Y_A"
@have "X - g''Y_B = X_A"
@have "g''Y_B = X_B"
@let "f' = func\_restrict\_image(func\_restrict(f, X_A))"
@let "g' = func\_restrict\_image(func\_restrict(g, Y_B))"
@have "\text{glue\_function2}(f', \text{inverse}(g')) \in (X_A \cup X_B) \cong (Y_A \cup Y_B)"
@qed

By comparison, the proof of Schroeder-Bernstein in Isabelle/ZF (as of Isabelle2017) spreads over 3 lemmas (Banach\_last\_equation, decomposition, schroeder\_bernstein), with 9 tactic invocations in total. The corresponding proof in Mizar (Th10 in knaster.miz) has 16 intermediate steps and is 39 lines long.

5.2 Rempe-Gillen’s Challenge

For our second example, we present our solution to a challenge problem proposed by Lasse Rempe-Gillen in a mailing list discussion\(^2\). See [2] for proofs of the same result in several other systems. The statement to be proved is:

\textbf{Lemma 1} Let $f$ be a continuous real-valued function on the real line, such that $f(x) > x$ for all $x$. Let $x_0$ be a real number, and define the sequence $x_n$ recursively by $x_{n+1} := f(x_n)$. Then $x_n$ diverges to infinity.

\(^2\) http://www.cs.nyu.edu/pipermail/fom/2014-October/018243.html
Our solution is as follows. We make use of several previously proved results: any bounded increasing sequence in \( \mathbb{R} \) converges (line 3), a continuous function \( f \) maps a sequence converging to \( x \) to a sequence converging to \( f \leftarrow x \) (line 5), and finally that the limit of a sequence in \( \mathbb{R} \) is unique.

**Lemma** rempe_gillen_challenge:
"real_fun(f) \implies \text{continuous}(f) \implies \text{incr_arg_fun}(f) \implies x_0 \in \mathbb{R} \implies S = \text{Seq}(\mathbb{R}, \lambda n. \text{nfold}(f,n,x_0)) \implies \neg \text{upper_bounded}(S)"

@proof
@have "seq_incr(S)"
@have "\forall n \in \mathbb{N}. S^\leftarrow n(x) \leq S^\leftarrow n(y)"
@obtain x where "\text{converges_to}(S,x)"
@let "T = \text{Seq}(\mathbb{R}, \lambda n. f^\leftarrow (S^\leftarrow n))"
@have "\text{converges_to}(T,f^\leftarrow x)"
@have "\text{converges_to}(T,x)"
@with
@have "\forall r > R_0 \exists k \in \mathbb{N}. \forall n \geq N_k. |T^\leftarrow n(x) - x| < R_r"
@obtain "k \in \mathbb{N}" where "\forall n \geq N_k. |S^\leftarrow n(x) - x| < R_r"
@have "\forall n \geq N_k. |T^\leftarrow n(x) - x| < R_r"
@with @have "T^\leftarrow n = S^\leftarrow (n + \mathbb{N})" @end
@end
@end
@end
@qed

The proof has 9 intermediate goals in total. [2] provides a comparison with Sledgehammer in Isabelle. The corresponding proof in Isabelle is divided into 16 intermediate goals, of which Sledgehammer solved 11 and failed on 5.

### 5.3 Construction of the Fundamental Group

In this section, we describe our construction of the fundamental group. The main goal of this case study is to demonstrate the expressiveness of untyped set theory, as well as the ability of our framework to rapidly reach deep results in mathematics. We will focus on stating the definitions and main results without proof. The entire formalization including proofs is 884 lines long.

Let \( I \) be the interval \([0, 1]\), equipped with the subspace topology from the topology on \( \mathbb{R} \):

**Definition** "\( I = \text{subspace}(\mathbb{R}, \text{closed_interval}(\mathbb{R}, 0_\mathbb{R}, 1_\mathbb{R})) \)"

Given two continuous maps \( f \) and \( g \) from \( S \) to \( T \), a homotopy between \( f \) and \( g \) is a continuous map from the product topology on \( S \times I \) to \( T \) that restricts to \( f \) and \( g \) at \( S \times \{0\} \) and \( S \times \{1\} \), respectively:

**Definition** is_homotopy :: "\([i, i, i] \Rightarrow o\) where

\[ \text{is_homotopy}(f,g,F) \leftrightarrow (\text{let } S = \text{source_str}(f) \text{ in let } T = \text{target_str}(f) \text{ in continuous}(f) \wedge \text{continuous}(g) \wedge S = \text{source_str}(g) \wedge T = \text{target_str}(g) \wedge F \in S \times T \rightarrow I \rightarrow T \wedge (\forall x \in S. \ F^\leftarrow(x,0_\mathbb{R}) = f^\leftarrow x \wedge F^\leftarrow(x,1_\mathbb{R}) = g^\leftarrow x)) \]

A path is a continuous function from \( I \). A homotopy between two paths is a path homotopy if it remains constant on \( \{0\} \times I \) and \( \{1\} \times I \):

**Definition** is_path :: "\( i \Rightarrow o \) where

\[ \text{is_path}(f) \leftrightarrow (f \in I \rightarrow T \text{ target_str}(f)) \]

**Definition** is_path_homotopy :: "\([i, i] \Rightarrow o\) where

\[ \text{is_path_homotopy}(f,g,F) \leftrightarrow (\text{is_path}(f) \wedge \text{is_path}(g) \wedge \text{is_homotopy}(f,g,F) \wedge (\forall t \in I. \ F^\leftarrow(0_\mathbb{R},t) = f^\leftarrow(0_\mathbb{R}) \wedge F^\leftarrow(1_\mathbb{R},t) = f^\leftarrow(1_\mathbb{R}))) \)"

\[ \text{Springer} \]
Two paths are path-homotopic if there exists a path homotopy between them. This is an equivalence relation on paths.

definition path_homotopic :: "i ⇒ i ⇒ o" where
  "path_homotopic(f,g) ←→ (∃F. is_path_homotopy(f,g,F))"

The path product is defined by gluing two morphisms. It is continuous by the pasting lemma:

definition I1 = subspace(\(\mathbb{R}\), closed_interval(\(\mathbb{R}\),0_\(\mathbb{R}\),1_\(\mathbb{R}\)/\(\mathbb{R}\) 2_\(\mathbb{R}\)))
definition I2 = subspace(\(\mathbb{R}\), closed_interval(\(\mathbb{R}\),1_\(\mathbb{R}\)/\(\mathbb{R}\) 2_\(\mathbb{R}\),1_\(\mathbb{R}\)))
definition interval_lower = Mor(I1,I,λ t. 2_\(\mathbb{R}\)∗R t)
definition interval_upper = Mor(I2,I,λ t. 2_\(\mathbb{R}\)∗R t −R 1_\(\mathbb{R}\))
definition path_product :: "i ⇒ i ⇒ i" (infixl "⋆" 70) where
  "f ⋆ g = glue_morphism(I, f ◦ m interval_lower, g ◦ m interval_upper)"

The loop space is a set of loops on \(X\). Path homotopy gives an equivalence relation on the loop space, and we define loop_classes to be the quotient set:

definition loop_space :: "i ⇒ i ⇒ i" where
  "loop_space(X,x) = {f ∈ I ↦ T X. f ′(0_\(\mathbb{R}\))=x ∧ f ′(1_\(\mathbb{R}\)) = x}"
definition loop_space_rel :: "i ⇒ i ⇒ i" where
  "loop_space_rel(X,x) = Equiv(loop_space(X,x), λfg . path_homotopic(f,g))"
definition loop_classes :: "i ⇒ i ⇒ i" where
  "loop_classes(X,x) = loop_space(X,x) // loop_space_rel(X,x)"

Finally, the fundamental group is defined as:

definition fundamental_group :: "i ⇒ i ⇒ i" ("\(\pi_1\)") where
  "\(\pi_1\)(X,x) = (let \(\mathcal{R}\) = loop_space_rel(X,x) in
  Group(loop_classes(X,x), equiv_class(\(\mathcal{R}\),const_mor(I,X,x)),
  λf g. equiv_class(\(\mathcal{R}\),rep(\(\mathcal{R}\),f) ⋆ rep(\(\mathcal{R}\),g))))"

To show that the fundamental group is actually a group, we need to show that the path product respects the equivalence relation given by path homotopy, and is associative up to equivalence (along with properties about inverse and identity). The final result is:

lemma fundamental_group_is_group:
  "is_top_space(X)⇒ x ∈ X⇒ is_group(\(\pi_1\)(X,x))"

An important property of the fundamental group is that a continuous function between topological spaces induces a homomorphism between their fundamental groups. This is defined as follows:

definition induced_mor :: "i ⇒ i ⇒ i" where
  "induced_mor(k,x) = (let X = source_str(k) in let Y = target_str(k) in
  let \(\mathcal{R}\) = loop_space_rel(X,x) in let \(\mathcal{S}\) = loop_space_rel(Y,k ′x) in
  Mor(\(\pi_1\)(X,x), \(\pi_1\)(Y,k ′x), λf g. equiv_class(\(\mathcal{S}\),k ◦ m rep(\(\mathcal{R}\),f))))"

The induced map is a homomorphism satisfying functorial properties:

lemma induced_mor_is_homomorphism:
  "continuous(k)⇒ X = source_str(k)⇒ Y = target_str(k)⇒ x ∈ source(k)⇒ induced_mor(k,x) ∈ \(\pi_1\)(X,x)⇒ g \(\pi_1\)(Y,k ′x)"

lemma induced_mor_id:
  "is_top_space(X)⇒ x ∈ X⇒ induced_mor(id_mor(X),x) = id_mor(\(\pi_1\)(X,x))"

lemma induced_mor_comp:
  "continuous(k)⇒ continuous(h)⇒ target_str(k) = source_str(h)⇒ x ∈ source(k)⇒ induced_mor(h ◦ m k, x) = induced_mor(h, k ′x) ◦ m induced_mor(k, x)"
5.4 Discussion

In this section, we discuss performance and usability issues encountered during our work.

The running time of auto2 remains a persistent issue. On a laptop with two 2.0GHz cores, the entire library of about 13,000 lines can be processed in about 20 minutes. This corresponds to about 1 second for every 10 lines of theory files. While still manageable for the current development, the performance is still worse in general compared to the use of regular Isabelle tactics (for example, the HOL-Analysis library, consisting of about 140,000 lines of code, can be processed on the same computer in about 13 minutes). This can be mainly attributed to the fact that auto2 performs more search at each step, and maintains more complex data structures, in particular the congruence closure.

The performance issue will need to be addressed in order to scale auto2 to larger and deeper formalizations. Improvements can be made in two directions. In one direction, the efficiency of the prover itself can be improved, in particular improving the incremental E-matching on the congruence closure. In the other direction, we would like to avoid repeating the proof search whenever possible. This could take the form of recording the result of the search in the theory file, in a form that can be quickly replayed as long as the proof remained valid.

Another aspect of usability is debugging a failed proof. When a proof fails, the most straightforward thing to try next is to break up the current subgoal into smaller subgoals. This can be done in an interactive way, as the subgoals are processed one-by-one, and auto2 will flag the first subgoal that it cannot prove. In some cases, however, the subgoal will appear very obvious, and it is desirable to understand why auto2 is unable to prove it. Currently, it is possible to let auto2 output a trace of its actions during the proof attempt. This includes the list of facts it derived, their dependencies, and proof steps used to derive them. However, there is no easy way to view the state of the rewrite table, property table, and the well-formedness table at the end of the proof attempt. Such user-interface improvements are possibilities of future work.

6 Related Work

In Isabelle, the main library for formalized mathematics using FOL is Isabelle/ZF. The basics of Isabelle/ZF is described in [15,16]. We also point to [15] for a review of earlier work on set theory from automated deduction and artificial intelligence communities. Outside the official Isabelle library, IsarMathLib [6] is a more recent project based on Isabelle/ZF. It formalized more results in abstract algebra and point-set topology, and also constructed the real numbers. The initial parts of our development closely parallels that in Isabelle/ZF, but we go further in several directions including constructing the number system. The primary difference in approach between our work and IsarMathLib is that we use auto2 for proofs, and develop our own system for handling structures, so that we do not make use of Isabelle tactics, Isar, or locales.

Outside Isabelle, the major formalization projects using set theory include Metamath [12] and Mizar [5], both of which have extensive mathematical libraries. There are some recent efforts to reproduce the Mizar environment in HOL-type systems [7,9]. While there are some similarities between our framework and Mizar’s, we do not aim for an exact reproduction. In particular, we maintain the typical style of stating definitions and theorems in Isabelle. More comparisons between our approach and Mizar can be found in Sect. 3.5.
Mizar formalized not just the definition of the fundamental group [8], but several of its properties, including the computation of the fundamental group of the circle. There is also a formalization of path homotopy in HOL Light which is then ported to Isabelle/HOL. This is used for the proof of the Brouwer fixed-point theorem and the Cauchy integral theorem, although the fundamental group itself does not appear to be constructed. The concept of fundamental groupoid, from which the fundamental group can be constructed, is also formalized in the ProofPower system [1].

In homotopy type theory, one can work with fundamental groups (and higher-homotopy groups) using synthetic definitions. This has led to formalizations of results about homotopy groups that are well beyond what can be achieved today using standard definitions (see [4] for a more recent example). We emphasize that our definition of the fundamental group, as with Mizar’s, follows the standard one in set theory.

7 Conclusion

We applied the auto2 prover to the formalization of mathematics using untyped set theory. Starting from the axioms of set theory, we formalized the definition of the fundamental group, as well as many other results in set theory, group theory, point-set topology, and real analysis. The entire development consists of over 13,000 lines of theory files and 3,500 lines of ML code, taking the author about 5 months to complete. Through this work, we demonstrated the ability of auto2 to scale to relatively large projects. We also hope this result can bring renewed interest to formalizing mathematics in untyped set theory in Isabelle.

Acknowledgements The author would like to thank the anonymous reviewers for their helpful comments. Most of this work is completed while the author is at the Department of Mathematics at MIT, and supported by NSF Award No. 1400713. Part of the work is done at the Technical University of Munich, while the author is supported by DFG Grant NI 491/16-1.

References

1. Arthan, R.D.: Some Mathematical Case Studies in ProofPower-HOL. https://www.lemma-one.com/papers/51.pdf. Accessed 22 July 2018
2. Blanchette, J.C., Kaliszyk, C., Paulson, L.C., Urban, J.: Hammering towards QED. J. Formaliz. Reason. 9(1), 101–148 (2016)
3. Bourbaki, N.: Theory of Sets. Springer, Berlin (2000)
4. Brunerie, G.: On the homotopy groups of spheres in homotopy type theory. Ph.D. Thesis. https://arxiv.org/abs/1606.05916
5. Grabowski, A., Kornilowicz, A., Naumowicz, A.: Mizar in a nutshell. J. Formaliz. Reason. Spec. Issue: User Tutor. 13(2), 153–245 (2010)
6. IsarMathLib: http://www.mongnu.org/isarmathlib/
7. Kaliszyk C., Pak, K., and Urban, J.: Towards a Mizar environment for Isabelle: foundations and language. In: Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2016), ACM, New York, NY, USA, pp. 58–65 (2016)
8. Kornilowicz, A., Shidama, Y., Grabowski, A.: The fundamental group. Formaliz. Math. 12(3), 261–268 (2004)
9. Kuncar, O.: Reconstruction of the Mizar type system in the HOL Light system. In: Pavlu, J., Safrankova, J. (eds.) WDS Proceedings of Contributed Papers: Part I—Mathematics and Computer Sciences, pp. 7–12. Matfyzpress (2010)
10. Lee, G., Rudnicki, P.: Alternative Aggregates in Mizar. In: Kauers M., Kerber M., Miner R., Windsteiger W. (eds) Towards mechanized mathematical assistants. LNCS, vol 4573, Springer, Berlin (2007)
11. Mahboubi, A., Tassi, E.: Canonical Structures for the Working Coq User. In: Blazy S., Paulin-Mohring C., Pichardie D. (eds.) ITP 2013. LNCS, vol 7998, Springer, Berlin, pp. 19–34 (2013)
12. Megill, N.D.: Metamath: A computer language for pure mathematics. http://us.metamath.org/downloads/metamath.pdf. Accessed 22 July 2018
13. Moura, L., Björner, N.: Efficient E-matching for SMT Solvers. In: Automated Deduction, CADE-21, LNCS 4603, pp. 183–198 (2007)
14. Munkres, J.R.: Topology. Prentice Hall, Upper Saddle River (2000)
15. Paulson, L.C.: Set theory for verification: I. From foundations to functions. J. Autom. Reason. 11(3), 353–389 (1993)
16. Paulson, L.C.: Set theory for verification: II. Induction and recursion. J. Autom. Reason. 15(2), 167–215 (1995)
17. Trybulec, A.: Some Features of the Mizar Language. In: ESPRIT Workshop (1993)
18. Wiedijk, F.: Mizar’s Soft Type System. In: Schneider K., Brandt J. (eds.) Theorem Proving in Higher Order Logics, LNCS, vol 4732, Springer, Berlin (2007)
19. Zhan, B.: AUTO2: a saturation-based heuristic prover for higher-order logic. In: Blanchette, J.C., Merz, S. (eds.): ITP 2016, LNCS 9807, pp. 441–456 (2016)