Isabelle/HOL/GST: A Formal Proof Environment for Generalized Set Theories

Isabelle/HOL/GST: A Formal Proof Environment for Generalized Set Theories*,**

Ciarán Dunne and J. B. Wells

Heriot-Watt University
https://www.macs.hw.ac.uk/ultra/

Abstract. A *generalized set theory* (GST) is like a standard set theory but also can have non-set structured objects that can contain other structured objects including sets. This paper presents Isabelle/HOL support for GSTs, which are treated as type classes that combine *features* that specify kinds of mathematical objects, e.g., sets, ordinal numbers, functions, etc. GSTs can have an exception feature that eases representing partial functions and undefinedness. When assembling a GST, extra axioms are generated following a user-modifiable policy to fill specification gaps. Specialized type-like predicates called *soft types* are used extensively. Although a GST can be used without a model, for confidence in its consistency we build a model for each GST from components that specify each feature's contribution to each tier of a von-Neumann-style cumulative hierarchy defined via ordinal recursion, and we then connect the model to a separate type which the GST occupies.

Keywords: Set theory · Higher-order logic · Soft types · Isabelle

1 Introduction

1.1 Set Theory. Many mathematicians (but not all) have long regarded set theories as suitable foundations of mathematics, in particular Zermelo/Fraenkel

* The version of this document published by Springer in the CICM 2022 proceedings can be found at https://doi.org/10.1007/978-3-031-16681-5_3. This document differs from the Springer version in the following ways. (1) A snapshot of the Isabelle source code corresponding to the tree of git commit e0ccdde0105eac05f3f4bbccdd58a9860e642eca dated 2022-06-03 12:15:29 +0100 is embedded in this PDF file as an attachment. (2) Two minor references to the ordered pair feature were removed. (We had failed to completely delete all traces of this feature from the paper that Springer published. The ordered pair feature can be found in the Isabelle source code and in the long version of this paper.) (3) It was clarified that the use of ∘ in the names of feature default parameters is just a convention and not a rule. (4) Minor issues with grammar, punctuation, cross-references, and spacing in formulas were fixed. (5) We did not implement Springer's different style of presenting the bibliography and other changes they made after we sent them the LaTeX source code to enforce their document style.

** Supported by EPSRC [EP/R513040/1 2273715].

set theory (ZF) and other related theories, e.g., ZF plus the Axiom of Choice (ZFC) and Tarski/Grothendieck (TG) set theory which adds a universe axiom. At its core, ZF is a domain type V whose members are called "sets", a predicate $\in$ (membership) of type $V \Rightarrow V \Rightarrow \mathsf{bool}$, and axioms specifying $\in$. Due to cardinality constraints, some operators like $\in$ can not themselves be members of the domain V but must live in higher types. Although just the predicate $\in$ is enough, formalizing ZF is easier with some constants and additional operators at a few further types, e.g., $\mathcal{P}$ (power set) and $\bigcup$ (union) at type $V \Rightarrow V$ and Repl (replacement) at type $V \Rightarrow (V \Rightarrow V \Rightarrow \mathsf{bool}) \Rightarrow V$. Nonetheless, ZF needs few types and nearly all interesting mathematical objects live in type V.

ZF is usually specified by "axioms" written in first-order logic (FOL), but ZF can also be given in higher-order logic (HOL). Actively used proof systems implementing ZF or TG in FOL include Isabelle/ZF, Mizar (TG), and Metamath (ZF, when using the set.mm database). Active systems in HOL include the Isabelle/HOL development "ZFC in HOL" [13]. Other systems include: Isabelle/Set [9] (TG, HOL), Isabelle/Mizar [8] (TG), and Egal [3] (TG, HOL).

In all these systems, except for a few key operators like $\in$, every mathematical object is a set. Arrangements of sets represent numbers, ordered pairs, functions, and nearly all other kinds of mathematical objects. Because sets are used to represent everything, representation overlaps are unavoidable. With the most commonly used representations, sad coincidences include, e.g., that $\langle 0, 1\rangle = \{1, 2\}$ where 0, 1, and 2 are natural numbers is a true equality between an ordered pair and a set, and that the successor function on natural numbers is equal to the number 1 as an integer. This troubles philosophers, leads university teachers to choose to deceive students, makes correct definitions more challenging, and adds difficulty to formalization.

1.2 Higher-Order Logic. Although many mathematicians favor set theory as a foundation of mathematics, many computer system implementers prefer formalisms where numerous types are used, rather than set theory's "one big type". Most of these systems extend Church's λ-calculus with simple types. We consider here HOL, which needs at its core only one type constructor $\Rightarrow$ and axioms and inference rules for constants for equality ($=$) and implication ($\rightarrow$). Most connectives ($\wedge$, $\neg$, $\forall$, etc.) are added via simple definitions that provide convenience, compactness, and readability, but no extra power. Sometimes domain-specific axioms are added that do provide extra power (which raises the question of whether these extensions are consistent, which we will address for our systems later in this paper). The systems we present in this paper are developed in Isabelle/HOL, the most active and widely used HOL proof system. Isabelle/HOL adds locale and type class mechanisms that support modularity, some limited type polymorphism, and overloading, and adds type definition features such as semantic subtypes, quotient types, and (co)recursive datatypes. Although Isabelle/HOL can be used just as a logical framework in which to define a set theory (as this paper does), typical uses of Isabelle/HOL generally put different kinds of mathematical objects into numerous fine-grained types.

1.3 Types. Consider ways "types" can be useful. One view of "type" is as an aspect of some object that can be inspected to help determine what you *should or want to* do with it. For example, when storing an incoming parcel, you might store food somewhere cold and jewelry in a safe. This corresponds to using types for overloading. Another view of "type" is as determining what you *meaningfully can* do with some object. For example, if you have two pieces of paper you want to attach, ice cream that you want to eat, a stapler, some staples, a bowl, and a spoon, you will want to use the right tools for each task. This corresponds to using types for avoiding meaningless combinations. There is no firm boundary between these two notions, e.g., you might *want* to store food somewhere cold because otherwise it might rot making the storage *not meaningful*.

Formal proving needs both views of "types" and these views interact. Using a lemma usually requires knowing if an operation is "defined" and the result's "type", and the answers to the same questions for operations on results *ad infinitum*. It is also frequently very useful to make decisions based on "types".

With this in mind, compare the set theory and HOL approaches (excluding HOL used solely as a framework for a set theory). HOL typically has precise and (usually) useful types for each object, obtained automatically with (often) reasonable efficiency, but sometimes it is quite hard to find types that allow fitting everything that is needed together. In set theory, typically the type of nearly everything is set, which conveys little information, but sets are a rich collection of every imaginable predicate and are usable as "types". However, there is (with currently available systems) a lack of automation for finding the right "types" and making them available in the right places at the right times.

1.4 Generalized Set Theories. A *generalized set theory* (GST) [1] specifies a domain type that contains set objects and may also contain other kinds of non-set structured objects, e.g., non-set ordered-pairs, non-set functions, non-set relations, etc. Many mathematicians view their work as based on ZF but their practice is often inconsistent with numbers, tuples, functions, etc., actually being sets. We believe much mathematics is, in effect, actually using a GST.

Subtype distinctions within a GST's domain type avoid representation overlaps. In a GST with non-overlapping sets, numbers, and ordered pairs, the answer to "is the ordered pair $\langle 0, 1 \rangle$ equal to the set $\{1, 2\}$?" is "no", an improvement over "yes" or "maybe" or "how dare you ask that question!". Subtype distinctions also help with showing that operations have meaningful results, and with choosing the most appropriate reasoning for each object.

A GST can have an exception object $\bullet$ (spoken "boom") which can not be confused with any other kind of object, can not lurk hidden inside any other object, and is useful for representing "undefined" results. This is similar to how terms in free logic can be "undefined" or fail to denote, but avoids the costs of actual undefinedness, e.g., inability to use standard HOL proof systems. This also avoids problems of other undefinedness approaches, e.g., the weakness of three-valued logic, the confusion caused by defining $1/0$ to be 0, the need for amazingly complicated function domain specifications to handle definedness gaps, etc. Our approach avoids needing separate inner/outer domain quantifiers.

Recent unformalized theoretical work by us with Kamareddine [5] showed how to combine *features* to make a GST. Features include what are traditionally called *structures*, e.g., the natural numbers, and also include "large" (proper-class-sized) concepts like the sets, the ordered pairs, the functions, and the ordinals. This approach, also followed in this paper, is as follows. Each GST has a single domain type `d` and some features. Each feature specifies the existence of some objects in `d`, which the feature is usually considered to "own", and can specify relationships among all the objects in `d`. Additional axioms specify that every object is "owned" by exactly one feature. Features are specified as independently as is feasible so, e.g., the specification of ordered pairs knows nothing about sets and *vice versa*. This is useful because, e.g., for sets and ordered pairs, (1) mathematicians think of these as distinct kinds of things, (2) the independent specification of ordered pairs is easier to comprehend than, e.g., Kuratowski's definition in terms of sets, and (3) this enables custom GSTs with only the desired primitive features. Our recent work also suggested theoretically how to build a model of a GST Q within another GST P and then connect that model to a type containing the GST Q, thereby reducing the question of the consistency of Q to the consistency of P. Our previous work had not, until this paper, been formalized.[1] Aside from this paper and the work mentioned above, the only model building for GSTs seems to be by Aczel and Lunnon [2], and their set theories have the anti-foundation axiom, which is quite different from our work.

1.5 Isabelle/HOL/GST. Building on Isabelle/HOL and earlier work on GSTs, we present a formal proving environment for defining GSTs and reasoning about them, including building models for them. The development consists of much Isabelle/Isar code for high-level formalization, as well as Isabelle/ML for the implementation of machinery that assists GST specification and model building. The source code of our development and instructions for use are available at `https://www.macs.hw.ac.uk/~cmd1/isabelle-gst/`.

Our development provides: (1) Definitions of a number of *soft type constructors*, operators that construct and manipulate *soft types* which are HOL predicates of type $\tau \Rightarrow \star$, where $\star$ is this paper's name for `bool`, and proofs of many useful properties of these. (2) A formal notion of *feature*, with a corresponding implementation as an Isabelle/ML record type. A feature contains a pointer to an Isabelle (type) class, which manages an abstract specification of the dependencies (on other classes/features), signature, axioms, definitions, theorems, syntax, etc., of a feature, and also contains information used when combining features. (3) Definitions and theorems in Isabelle theories for features for ZF-style sets, functions, ordinals, ordinal recursion, an exception object, ordered pairs, natural numbers and binary relations.[2] (4) Automation for combining features to define GSTs, which are implemented as Isabelle classes with axioms for

[1] A mostly formalized proof had been given earlier in Isabelle/ZF of the existence of a model of a much simpler system with just two fixed features [4].

[2] Natural numbers and binary relations can be found in the Isabelle source code, as can ordered pairs, which also have a LaTeX presentation in this paper's long version.

enforcing: (a) that each object in the domain of individuals is owned by exactly one feature, (b) a policy specifying allowed combinations of objects of different features, and a policy for filling in specification gaps. These two policies primarily support approaches to handling "undefined" operations, including the use of an exception object. (5) Generic definitions and reasoning for GST models, which are cumulative hierarchies defined using ordinal recursion. (6) A notion of *model component*, with a corresponding implementation as an ML record type. Each model component specifies a schematic description of a part of a model, primarily constraints placed on the zero, successor, and limit cases of the ordinal recursion used in defining models. (7) Automatic generation of terms, theorems and proof states that assist in implementing GST models. (8) Automatic lifting and generation of transfer rules for constants on types obtained by Isabelle/HOL type definitions on GST models, gained by interfacing Lifting and Transfer. (9) A bootstrap of our development from Paulson's "ZFC in HOL", carried out by instantiating our GZF (set), Ordinal, OrdinalRec (ordinal recursion), OPair (ordered pair), and Function classes at the type $\mathbf{V}$ of "ZFC in HOL", which makes a GST in which every object is a set. (10) An example GST called ZF^+ (defined as a class) with sets and all of the following as non-set objects: functions, ordinals, and the exception object. (11) An Isabelle type d_0 which instantiates the ZF^+ class, obtained by type definition on a model built using our model-building kit in $\mathbf{V}$, justifying confidence in using ZF^+, provided you have faith in Isabelle/HOL and the axioms of ZFC that were added at $\mathbf{V}$.

Items (5), (6), (7), and (8) with model building details are not presented in this paper, but are described in the appendix of the long version of this paper. Full details can be found in the `ModelKit/` directory within the Isabelle source.

1.6 Summary of Contributions. This paper presents a formalization of generalized set theories in Isabelle/HOL. Section 2 formulates a logical framework with classes like those of Isabelle. Section 3.1 defines GST features as classes with associated data. Section 3.2 presents example features for ZF-style sets, ordinals, functions, and an exception element. Section 3.3 defines how to combine features to create a GST and shows how to combine our example features to make the GST ZF^+ which has all these features within one domain type. Section 4 presents examples of working in ZF^+. Section 5 presents a methodology for building models of GSTs in Isabelle and how we use the Lifting and Transfer packages to create types that instantiate GSTs. Section 5 also discusses how we justify confidence in ZF^+ by building a model in the type $\mathbf{V}$ of "ZFC in HOL", defining a type d_0 isomorphic to this model, and instantiating ZF^+ at d_0.

2 Mathematical Definitions and Logical Framework

We assume a set-theoretic meta-level with at least: equality $\equiv$; definitions $:=$; an empty set $\varnothing$; set membership ϵ, binary union $\Cup$, set literals $\{\!|\, \square_1, \ldots, \square_n \,|\!\}$, and comprehensions $\{\!|\, \square \mid \square \,|\!\}$; an empty list $\diamond$, tuple/list literals $[\square_1, \ldots, \square_n]$, membership ϵ:, cons $\#$, append @, and comprehensions $[\,\square \mid \square\,]$; and a set of

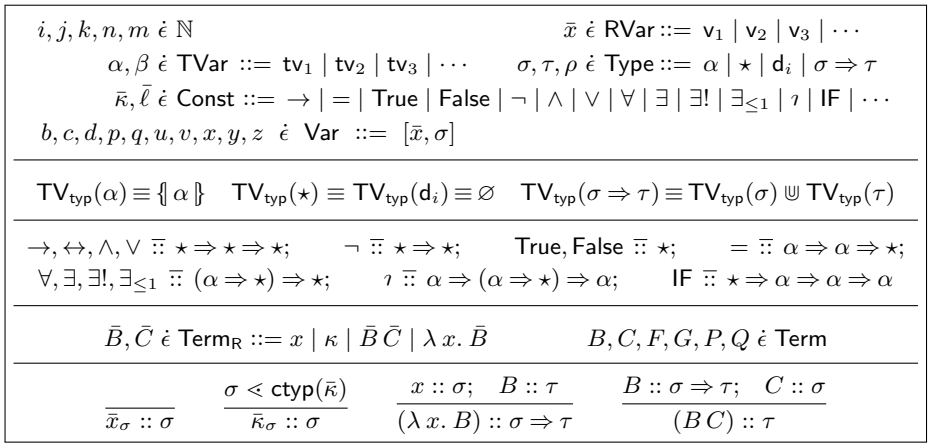

Fig. 1. Base syntax, type variables, constants, terms, and the typing relation.

natural numbers $\mathbb{N}$. Uses of $\mathrel{\dot\in}$ declare that symbols on the left side of $\mathrel{\dot\in}$ are metavariables ranging over the set on the right side.

2.1 Syntax and Types. Our logical framework is close to and inspired by the Isabelle/HOL formulation of Kunčar and Popescu [12]. Figure 1 defines the meta-level sets TVar of *type variables*, RVar of *raw term variables*, and Type, Var, and Const. Type consists of type variables, the type $\star$ of truth claims/assumptions, *domain types* d_i, and *operator types* $\sigma \Rightarrow \tau$.[3] The constructor $\Rightarrow$ is right associative so $(\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3) \equiv (\tau_1 \Rightarrow (\tau_2 \Rightarrow \tau_3))$. $\mathsf{TV}_{\mathsf{typ}}(\sigma)$ yields the set of type variables occurring in σ. Var is the set of *term variables*, which are pairs of raw term variables and types. A term variable $[\bar{x}, \sigma]$ may be written as $\bar{x}_\sigma$. Const is a set of *constants*. The metavariable $\mathcal{K}$ ranges over lists of constants. The fixed meta-level function $\mathsf{ctyp} : \mathsf{Const} \to \mathsf{Type}$ assigns each constant a type. We write $\bar{\kappa} \mathrel{::} \tau$ for $\mathsf{ctyp}(\bar{\kappa}) \equiv \tau$. Figure 1 assigns some constants some types.

The notation $\sigma[\alpha := \tau]$ denotes the *type substitution* replacing occurrences of the type variable α in σ by τ. A type σ is an *instance* of τ (written $\sigma \lessdot \tau$) iff $\sigma \equiv \tau[\alpha_1 := \rho] \ldots [\alpha_n := \rho_n]$ for some $\rho_1, \ldots, \rho_n$. For example, $(\mathsf{d}_1 \Rightarrow \mathsf{d}_1) \lessdot (\alpha \Rightarrow \alpha)$. A *constant instance* is a pair $[\bar{\kappa}, \sigma]$ such that $\sigma \lessdot \mathsf{ctyp}(\bar{\kappa})$ (e.g., $[\mathcal{P}, \mathsf{d}_0 \Rightarrow \mathsf{d}_0]$ and $[\in, \mathsf{d}_1 \Rightarrow \mathsf{d}_1]$ are constant instances for the constants $\mathcal{P} \mathrel{::} \alpha \Rightarrow \alpha$ and $\in \mathrel{::} \alpha \Rightarrow \alpha \Rightarrow \star$ respectively). Constant instances $[\bar{\kappa}, \sigma]$ may be written as $\bar{\kappa}_\sigma$, and if $\mathsf{ctyp}(\bar{\kappa})$ has no type variables, then we may write $\bar{\kappa}$ for the constant instance $\bar{\kappa}_{\mathsf{ctyp}(\bar{\kappa})}$. Let θ and κ range over constant instances.

Figure 1 defines the set $\mathsf{Term}_{\mathsf{R}}$ of *raw terms*. A raw term is a variable x, a constant instance κ, an *application* $\bar{B}\,\bar{C}$, or an *abstraction* $\lambda\, x.\, \bar{B}$. Define the *free variable* meta-level function $\mathsf{FV}_{\mathsf{trm}}$ and α-equivalence on $\mathsf{Term}_{\mathsf{R}}$ as usual. Let Term_α be $\mathsf{Term}_{\mathsf{R}}$ modulo α-equivalence. Define *term substitution* and *type substitution* on Term_α as usual with the notation $\bar{B}[x := \bar{C}]$ and $\bar{B}[\alpha := \sigma]$. Define

[3] At object level, "function" means an object satisfying the Fun predicate of figure 4.

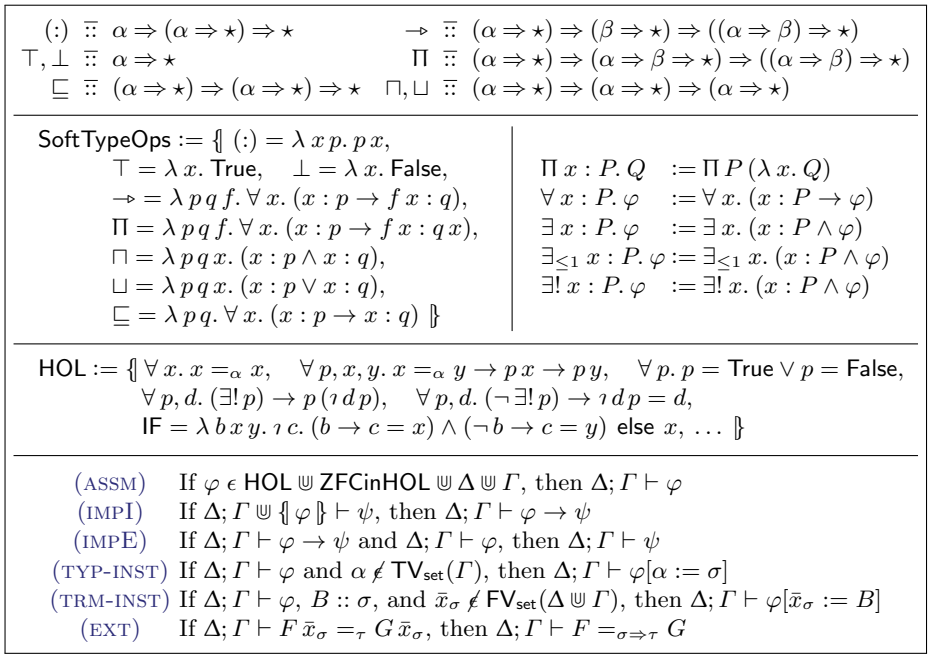

Fig. 2. Types, definitions and notation for soft types, axioms, and inference rules.

β-equivalence on Term_α as usual. Let Term be Term_α modulo β-equivalence. Lift $\mathsf{FV}_{\mathsf{trm}}$ and term and type substitution to Term.

Let the typing relation :: between Term and Type be the least relation satisfying the rules in figure 1. Let a term B be a *formula* iff $B :: \star$. A *simple definition* is a formula of the form $\kappa =_\sigma B$.[4] Let φ, ψ, and γ range over formulas, let Δ, Γ, and Θ range over sets of formulas, and let Φ range over formula lists.

We adopt the following notation. Given $x \equiv \bar{x}_\sigma$, the expression $\lambda\, x :: \sigma.\, B$ stands for $\lambda\, \bar{x}_\sigma.\, B$. Inside term expressions $B =_\sigma C$ and $\lambda\, x :: \sigma.\, B$, we allow omitting the type σ (and the ::), provided that σ can be uniquely determined by the typing rules and other type information in or about B and C. The notation $\lambda x_1 \cdots x_n.\, B$ stands for the nested abstractions $\lambda x_1.\, \cdots\, (\lambda x_n.\, B)$. The notation $[\kappa_1 :: \tau_1, \ldots, \kappa_n :: \tau_n]$ denotes the list $[\kappa_1, \ldots, \kappa_n]$ and asserts that $\kappa_i \in \mathsf{Const}$ and $\kappa_i :: \tau_i$ for $1 \leq i \leq n$.

If $\bar{\kappa}$ is *infix*, an application $(\bar{\kappa}_\sigma\, B)\, C$ is written as $B\, \bar{\kappa}_\sigma\, C$. The constants $=$, $\wedge$, $\vee$, and $\rightarrow$ are all infix, and listed here in descending order of precedence. Negation and operator application take precedence over infix operators, e.g., $\neg\, P \wedge \neg\, Q$ is $(\neg\, P) \wedge (\neg\, Q)$ and $F\, x =_\sigma G\, x$ is $(F\, x) =_\sigma (G\, x)$. If $\bar{\kappa}$ is a *binder*, an application $\bar{\kappa}_\sigma\, (\lambda x :: \sigma.\, B)$ is written as $(\bar{\kappa} x :: \sigma.\, B)$, and $\bar{\kappa}\, x_1, \ldots, x_n.\, B$ stands for the nested applications of quantifiers and abstractions $\bar{\kappa}_\sigma\, (\lambda\, x_1 :: \sigma.\, \cdots (\bar{\kappa}_\sigma\, (\lambda\, x_n :: \sigma.\, B)))$. The constants $\forall$, $\exists$, $\exists!$, and $\exists_{\leq 1}$ are all binders.

[4] B must not refer to κ, including via a chain of other definitions.

2.2 Soft Types. A *soft type* is an operator of type $\tau \Rightarrow \star$ for some τ. A key difference from "hard" types is that each object will satisfy many (often infinitely many) soft types. Figure 2 gives types and simple definitions for operators for building and using soft types. Our soft type constructors are mostly the same as those used by Kappelmann, Chen, and Krauss in Isabelle/Set [9], which in turn are along the lines suggested much earlier by Krauss [10]. To help the reader think "types", and also to allow proof tactics to follow soft-type-specific strategies, we write "$F : P$" as a *soft typing* which means the same thing as "$P\,F$", i.e., F satisfies the predicate P. We also have a non-dependent constructor $\twoheadrightarrow$ and a dependent constructor $\mathsf{\Pi}$ for soft types on operators, soft intersection and union type constructors $\sqcap$ and $\sqcup$, and soft subtyping $\sqsubseteq$. Our development derives the standard introduction and elimination rules for each of these concepts. The constants $(:)$, $\twoheadrightarrow$, $\sqcap$, $\sqcup$, and $\sqsubseteq$ are all infix, and $\twoheadrightarrow$ is right-associative. Figure 2 gives notation for dependent operator soft types and restricted quantification. Let $\mathsf{SoftTypeOps}$ be the set of simple definitions as defined in figure 2.

For example, using the soft operator type constructor $\twoheadrightarrow$, given a predicate $P :: \sigma \Rightarrow \star$ and a predicate $Q :: \tau \Rightarrow \star$, the term $P \twoheadrightarrow Q$ is a predicate of type $(\sigma \Rightarrow \tau) \Rightarrow \star$ such that if $P \twoheadrightarrow Q$ is true of $x :: \sigma \Rightarrow \tau$ and P is true of $y :: \sigma$ then Q is true of $x\,y :: \tau$. Precisely, $F : P \twoheadrightarrow Q$ means $\forall\, b.\ b : P \to F\,b : Q$.

2.3 Inference Rules and Axioms. Let $\mathsf{TV_{trm}}$ be the extension of $\mathsf{TV_{typ}}$ to terms. Let $\mathsf{FV_{set}}(\mathit{\Gamma})$ be the union of all $\mathsf{FV_{trm}}(\varphi)$ for all $\varphi \in \mathit{\Gamma}$. Let $\mathsf{TV_{set}}(\mathit{\Gamma})$ be the union of all $\mathsf{TV_{trm}}(\varphi)$ for all $\varphi \in \mathit{\Gamma}$. Let $\mathit{\Gamma}[x := B]$ be the set of all $\varphi[x := B]$ for all $\varphi \in \mathit{\Gamma}$.

HOL is the set of axioms (formulas) given in figure 2 that implement reflexivity of equality, indiscernability of equal objects, the law of the excluded middle, a definite description operator with a default ($\imath$), a conditional operator (IF), and simple definitions (not shown) for True, $\forall$, $\exists$, False, $\neg$, $\wedge$, $\vee$, $\leftrightarrow$, $\exists_{\leq 1}$, and $\exists!$. Nice notation for $\imath$ and IF are given thus: $(\imath\, x.\ \varphi\ \mathsf{else}\ D) := \imath\, D\, (\lambda\, x.\ \varphi)$, $(\mathsf{if}\ P\ \mathsf{then}\ B\ \mathsf{else}\ C) := \mathsf{IF}\, P\, B\, C$. Let $\mathsf{ZFCinHOL}$ be the set of axioms used in Paulson's "ZFC in HOL" [13]. Let the *deduction relation* $\vdash$ be the least relation satisfying the inference rules in figure 2. Our normal use will be to derive judgements of the form $\mathsf{HOL} \Cup \mathsf{SoftTypeOps} \Cup \Delta; \mathit{\Gamma} \vdash \varphi$ where Δ contains additional definitions specific to the topic of the proof and $\mathit{\Gamma}$ contains local assumptions.

2.4 Classes. A *(type) class* is a tuple $\mathcal{C} \equiv [\mathcal{D}, \mathcal{K}, \Phi, \Theta]$. $\mathcal{D}$ is a list of classes called the *dependencies* of $\mathcal{C}$. $\mathcal{K}$ is a list $[\kappa_1, \ldots, \kappa_n]$ of pairwise distinct constants called the *parameters* of $\mathcal{C}$, such that $\mathsf{TV_{trm}}(\kappa_1) \Cup \ldots \Cup \mathsf{TV_{trm}}(\kappa_n) \equiv \{\!|\, \alpha \,|\!\}$ for some α, i.e., exactly one type variable occurs in the parameters of $\mathcal{C}$, which we refer to as $\mathsf{tv}(\mathcal{C})$. Φ is a list of formulas and Θ is a list of simple definitions, called the *axioms* and *definitions* of $\mathcal{C}$ respectively.

We write $\sigma[\![\mathsf{d}_i]\!]$ and $B[\![\mathsf{d}_i]\!]$ for the results of the type substitutions $\sigma[\mathsf{tv}(\mathcal{C}) := \mathsf{d}_i]$ and $B[\mathsf{tv}(\mathcal{C}) := \mathsf{d}_i]$ respectively. A *parameter instantiation* of $\mathcal{C}$ at d_i is a set of formulas $\{\!|\, \kappa_1[\![\mathsf{d}_i]\!] = B_1, \ldots, \kappa_n[\![\mathsf{d}_i]\!] = B_n \,|\!\}$, where $\mathsf{TV}(B_j) \equiv \varnothing$ for $j \in \{\!|\, 1, \ldots, n \,|\!\}$. Relative to a set of hypothesis $\mathit{\Gamma}$ and a set of definitions Δ, we say

$$
\begin{array}{rl}
\mathsf{GZF}_{\mathsf{consts}} := & [\ \circ_{\mathsf{GZF}} :: \alpha,\ \mathsf{Set} :: \alpha \Rightarrow \star,\ \in :: \alpha \Rightarrow \alpha \Rightarrow \star,\ \bigcup :: \alpha \Rightarrow \alpha,\ \mathcal{P} :: \alpha \Rightarrow \alpha, \\
& \emptyset :: \alpha,\ \mathsf{Succ} :: \alpha \Rightarrow \alpha,\ \mathsf{Inf} :: \alpha,\ \mathsf{Repl} :: \alpha \Rightarrow (\alpha \Rightarrow \alpha \Rightarrow \star) \Rightarrow \alpha\ ] \\
\mathsf{GZF}_{\mathsf{axioms}} := & [\ \bigcup : \mathsf{SetOf}\,\mathsf{Set} \rightarrowtriangle \mathsf{Set},\quad \mathcal{P} : \mathsf{Set} \rightarrowtriangle \mathsf{SetOf}\,\mathsf{Set}, \\
& \emptyset : \mathsf{Set},\quad \mathsf{Succ} : \mathsf{Set} \rightarrowtriangle \mathsf{Set},\quad \mathsf{Inf} : \mathsf{Set}, \\
& \mathsf{Repl} : (\Pi\, x : \mathsf{Set}.\ \mathsf{ReplPred}\, x \rightarrowtriangle \mathsf{Set}), \\
& \forall\, x, y : \mathsf{Set}.\ \forall\, b.\ (b \in x \leftrightarrow b \in y) \rightarrow x = y, \\
& \forall\, x : \mathsf{SetOf}\,\mathsf{Set}.\ \forall\, b.\ b \in \bigcup x \leftrightarrow (\exists\, y.\ y \in x \wedge b \in y), \\
& \forall\, x, y : \mathsf{Set}.\ y \in \mathcal{P}\, x \leftrightarrow y \subseteq x, \\
& \forall\, b.\ \neg\, b \in \emptyset,\quad \forall\, x : \mathsf{Set}.\ b \in \mathsf{Succ}\, x \leftrightarrow (b \in x \vee b = x), \\
& \emptyset \in \mathsf{Inf} \wedge (\forall\, b.\ b \in \mathsf{Inf} \rightarrow \mathsf{Succ}\, b \in \mathsf{Inf}), \\
& \forall\, x : \mathsf{Set}.\ \forall\, p : \mathsf{ReplPred}\, x. \\
& \quad \forall\, c.\ c \in \mathsf{Repl}\, x\, p \leftrightarrow (\exists\, b.\ b \in x \wedge p\, b\, c \wedge c : \mathsf{SetMem})\ ] \\
\mathsf{GZF}_{\mathsf{defs}} := & [\ \subseteq\ = (\lambda\, x\, y.\ \forall\, b.\ b \in x \rightarrow b \in y),\quad \mathsf{SetMem} = (\lambda\, b.\ \exists\, y : \mathsf{Set}.\ b \in y), \\
& \mathsf{SetOf}\ = (\lambda\, p\, x.\ x : \mathsf{Set} \wedge \forall\, b \in x.\ b : p), \\
& \mathsf{ReplPred} = (\lambda\, x\, p.\ \forall\, b \in x.\ \exists_{\leq 1}\, c : \mathsf{SetMem}.\ p\, b\, c)\ ]
\end{array}
$$

$$
\begin{array}{rl}
\mathsf{Ord}_{\mathsf{consts}} := & [\ \circ_{\mathsf{Ord}} :: \alpha,\ \mathsf{Ord} :: \alpha \Rightarrow \star,\ < :: \alpha \Rightarrow \alpha \Rightarrow \star,\ 0 :: \alpha,\ \mathsf{succ} :: \alpha \Rightarrow \alpha,\ \omega :: \alpha\ ] \\
\mathsf{Ord}_{\mathsf{axioms}} := & [\ 0 : \mathsf{Ord},\quad \mathsf{succ} : \mathsf{Ord} \rightarrowtriangle \mathsf{Ord},\quad \omega : \mathsf{Limit}, \\
& \forall\, u : \mathsf{Ord}.\ \neg\, u < 0, \\
& \forall\, u, v : \mathsf{Ord}.\ u < \mathsf{succ}\, v \leftrightarrow (u < v \vee u = v), \\
& \forall\, u : \mathsf{Limit}.\ u = \omega \vee \omega < u, \\
& \forall\, u, v, w : \mathsf{Ord}.\ u < v \rightarrow v < w \rightarrow u < w \quad (\text{transitivity}), \\
& \forall\, u, v : \mathsf{Ord}.\ u < v \rightarrow \neg\, v < u \quad (\text{antisymmetry}), \\
& \forall\, u, v : \mathsf{Ord}.\ u < v \vee u = v \vee v < u \quad (\text{trichotomy}), \\
& \forall\, p.\ (\forall\, u : \mathsf{Ord}.\ (\forall\, v : \mathsf{Ord}.\ v < u \rightarrow p\, v) \rightarrow p\, u) \rightarrow (\forall\, w : \mathsf{Ord}.\ p\, w)\ ] \\
\mathsf{Ord}_{\mathsf{defs}} := & [\ \mathsf{Limit} = (\lambda\, u.\ u : \mathsf{Ord} \wedge 0 < u \wedge (\forall\, v : \mathsf{Ord}.\ v < u \rightarrow \mathsf{succ}\, v < u))\ ]
\end{array}
$$

$$
\begin{array}{rl}
\mathsf{OrdRec}_{\mathsf{consts}} := & [\ \circ_{\mathsf{OrdRec}} :: \alpha,\ \mathsf{predSet} :: \alpha \Rightarrow \alpha,\ \mathsf{supOrd} :: \alpha \Rightarrow \alpha, \\
& \mathsf{OrdRec} :: (\alpha \Rightarrow (\alpha \Rightarrow \alpha) \Rightarrow \alpha) \Rightarrow (\alpha \Rightarrow \alpha \Rightarrow \alpha) \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \alpha\ ] \\
\mathsf{OrdRec}_{\mathsf{axioms}} := & [\ \mathsf{predSet} : \mathsf{Ord} \rightarrowtriangle \mathsf{SetOf}\,\mathsf{Ord},\quad \mathsf{supOrd} : \mathsf{SetOf}\,\mathsf{Ord} \rightarrowtriangle \mathsf{Ord}, \\
& \forall\, u, v : \mathsf{Ord}.\ u \in \mathsf{predSet}\, v \leftrightarrow u < v, \\
& \forall\, x : \mathsf{SetOf}\,\mathsf{Ord}.\ \forall\, u.\ u \in x \rightarrow u < \mathsf{succ}\,(\mathsf{supOrd}\, x), \\
& \forall\, g, f, x.\ \mathsf{OrdRec}\, g\, f\, x\, 0 = x, \\
& \forall\, g, f, x.\ \forall\, u : \mathsf{Ord}.\ \mathsf{OrdRec}\, g\, f\, x\,(\mathsf{succ}\, u) = f\,(\mathsf{succ}\, u)\,(\mathsf{OrdRec}\, g\, f\, x\, u), \\
& \forall\, g, f, x.\ \forall\, u : \mathsf{Limit}.\ \mathsf{OrdRec}\, g\, f\, x\, u = \\
& \qquad g\, u\,(\lambda\, v.\ \mathsf{if}\ v : \mathsf{Ord} \wedge v < u\ \mathsf{then}\ \mathsf{OrdRec}\, g\, f\, x\, v\ \mathsf{else}\ \circ_{\mathsf{OrdRec}})\ ]
\end{array}
$$

Fig. 3. Constants, axioms, and definitions for the GZF, Ordinal, and OrdinalRec classes

that $\mathcal{C}$ is *instantiated* at d_i if we have $\Delta; \Gamma \vdash \varphi$ for any $\varphi \in \Phi[\![\mathsf{d}_i]\!]$. Typically Δ will contain parameter instantiations for all of the dependencies of $\mathcal{C}$.

3 GSTs as Type Classes

3.1 GST Features. A *feature* is a tuple $\mathcal{F} \equiv [\mathcal{C}, P_{\mathsf{logo}}, P_{\mathsf{cargo}}, \kappa_{\mathsf{default}}]$, where $\mathcal{C}$ is a class, $P_{\mathsf{logo}}, P_{\mathsf{cargo}} :: \mathsf{tv}(\mathcal{C}) \Rightarrow \star$ are the *cargo* and *logo* soft types of $\mathcal{F}$, and

$$
\begin{aligned}
\mathsf{Fun}_{\mathsf{consts}} &:= [\, \circ_{\mathsf{Fun}} :: \alpha,\ \mathsf{Fun} :: \alpha \Rightarrow \star,\ \mathsf{app} :: \alpha \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \star,\ \nrightarrow :: \alpha \Rightarrow \alpha \Rightarrow \alpha,\\
&\quad\ \mathsf{mkFun} :: \alpha \Rightarrow (\alpha \Rightarrow \alpha \Rightarrow \star) \Rightarrow \alpha,\ \mathsf{dom} :: \alpha \Rightarrow \alpha,\ \mathsf{ran} :: \alpha \Rightarrow \alpha \,]\\
\mathsf{Fun}_{\mathsf{axioms}} &:= [\, \mathsf{mkFun} : (\Pi\, x : \mathsf{Set}.\ \mathsf{FunPred}\, x \rightharpoonup \mathsf{Fun}),\\
&\quad\ \mathsf{dom} : \mathsf{Fun} \rightharpoonup \mathsf{Set},\ \mathsf{ran} : \mathsf{Fun} \rightharpoonup \mathsf{Set},\\
&\quad\ \nrightarrow : \mathsf{Set} \rightharpoonup \mathsf{Set} \rightharpoonup \mathsf{SetOf}\, \mathsf{Fun},\\
&\quad\ \forall\, f : \mathsf{Fun}.\ \forall\, b, c, d.\ \mathsf{app}\, f\, b\, c \wedge \mathsf{app}\, f\, b\, d \rightarrow c = d\\
&\quad\ \forall\, f, g : \mathsf{Fun}.\ (\forall\, b, c.\ \mathsf{app}\, f\, b\, c \leftrightarrow \mathsf{app}\, g\, b\, c) \rightarrow f = g\\
&\quad\ \forall\, f : \mathsf{Fun}.\ \forall\, b.\ b \in \mathsf{dom}\, f \leftrightarrow (\exists\, c.\ \mathsf{app}\, f\, b\, c)\\
&\quad\ \forall\, f : \mathsf{Fun}.\ \forall\, c.\ c \in \mathsf{ran}\, f \leftrightarrow (\exists\, b.\ \mathsf{app}\, f\, b\, c)\\
&\quad\ \forall\, x, y : \mathsf{Set}.\ \forall\, f : \mathsf{Fun}.\ (f \in x \nrightarrow y) \leftrightarrow (\mathsf{dom}\, f \subseteq x \wedge \mathsf{ran}\, f \subseteq y)\\
&\quad\ \forall\, x : \mathsf{Set}.\ \forall\, p : \mathsf{FunPred}\, x.\ \forall\, b, c.\\
&\qquad \mathsf{app}\, (\mathsf{mkFun}\, x\, p)\, b\, c \leftrightarrow (b \in x \wedge p\, b\, c \wedge b : \mathsf{FunMem} \wedge c : \mathsf{FunMem})\,]\\
\mathsf{Fun}_{\mathsf{defs}} &:= [\, \mathsf{FunMem} = (\lambda\, b.\ \exists\, f : \mathsf{Fun}.\ b \in \mathsf{dom}\, f \vee b \in \mathsf{ran}\, f),\\
&\quad\ \mathsf{FunPred} = (\lambda\, x\, p.\ \forall\, b : \mathsf{FunMem}.\ b \in x \rightarrow (\exists_{\leq 1}\, c : \mathsf{FunMem}.\ p\, b\, c))\,]
\end{aligned}
$$

Fig. 4. Constants, axioms, and definitions for the Function class

$$
\begin{aligned}
\mathsf{GZF} &:= [\, \diamond, \mathsf{GZF}_{\mathsf{consts}}, \mathsf{GZF}_{\mathsf{axioms}}, \mathsf{GZF}_{\mathsf{defs}}\,]\\
\mathsf{Ordinal} &:= [\, \diamond, \mathsf{Ord}_{\mathsf{consts}}, \mathsf{Ord}_{\mathsf{axioms}}, \mathsf{Ord}_{\mathsf{defs}}\,]\\
\mathsf{OrdinalRec} &:= [\, [\, \mathsf{GZF}, \mathsf{Ordinal}\,], \mathsf{OrdRec}_{\mathsf{consts}}, \mathsf{OrdRec}_{\mathsf{axioms}}, \mathsf{OrdRec}_{\mathsf{defs}}\,]\\
\mathsf{Function} &:= [\, [\, \mathsf{GZF}\,], \mathsf{Fun}_{\mathsf{consts}}, \mathsf{Fun}_{\mathsf{axioms}}, \mathsf{Fun}_{\mathsf{defs}}\,]\\
\mathsf{Exception} &:= [\, \diamond, [\, \circ_{\mathsf{Exc}} :: \alpha, \mathsf{Exc} :: \alpha \Rightarrow \star, \bullet :: \alpha\,], \diamond, \diamond\,]
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{GZF} &:= [\, \mathsf{GZF}, \mathsf{Set}, \mathsf{SetMem}, \circ_{\mathsf{GZF}}\,]\\
\mathbf{Ordinal} &:= [\, \mathsf{Ordinal}, \mathsf{Ord}, \bot, \circ_{\mathsf{Ord}}\,]\\
\mathbf{OrdRec} &:= [\, \mathsf{OrdinalRec}, \bot, \bot, \circ_{\mathsf{OrdRec}}\,]\\
\mathbf{Function} &:= [\, \mathsf{Function}, \mathsf{Fun}, \mathsf{FunMem}, \circ_{\mathsf{Fun}}\,]\\
\mathbf{Exc} &:= [\, \mathsf{Exception}, \mathsf{Exc}, \bot, \circ_{\mathsf{Exc}}\,]
\end{aligned}
$$

Fig. 5. Definition of our example features and their classes.

$\kappa_{\mathsf{default}}$ is a constant called the *default parameter* of $\mathcal{F}$. The terms P_{logo}, P_{cargo}, and $\kappa_{\mathsf{default}}$ keep track of information used when combining features to create GSTs. P_{logo} (a.k.a. $\mathsf{logo}(\mathcal{F})$) should be chosen as the soft type of all objects contributed to the domain by a feature's axioms. For features that do not contribute any objects to the domain, $\bot$ should be used. P_{cargo} (a.k.a. $\mathsf{cargo}(\mathcal{F})$) should be chosen as the soft type satisfied by all objects contained in the internal structure of some object $X : P_{\mathsf{logo}}$. Keeping track of cargo types allows preventing the exception object $\bullet$ from being contained in sets and functions, allowing the benefits of a free logic, i.e., terms can be "undefined", without the need for anything to actually really be undefined and without the need for separate quantifiers for an "inner" and "outer" domain. The object $\bullet$ is in the spirit of concepts like the number 0 and the empty set $\emptyset$, i.e., it is an object representing what would otherwise be the lack of an object. Each feature's default parameter $\kappa_{\mathsf{default}}$ must occur in the

parameter list of the feature's class $\mathcal{C}$.[5] The default parameter is intended to be a placeholder that can be used in a feature's axioms and definitions for exceptional results and normally it will be axiomatized to be equal to some specific object (typically $\bullet$) when features are combined into a GST.

3.2 Example Features. We now define features for sets, ordinals, ordinal recursion, functions, and an exception ($\bullet$). Figures 3 and 4 define constants, axioms, and definitions for each feature's class. Figure 5 defines the classes and their features (in boldface). The Isabelle/HOL development can be found in `GST_Features.thy`.

We have formulated some of our features' axioms as soft typings. This is more useful in a GST than in set-only ZF, because even basic set operations like $\bigcup$ and $\mathcal{P}$ can be "undefined" because, e.g., the argument might be a non-set.

We use a number of soft types that classify objects in a GST's domain, e.g., Set, Ord, and $\mathsf{Function}$. For example, $\emptyset : \mathsf{Set}$. An example of using the soft operator type constructor $\rightarrowtriangle$ is that using the axioms $\mathsf{dom} : \mathsf{Fun} \rightarrowtriangle \mathsf{Set}$ and $\mathcal{P} : \mathsf{Set} \rightarrowtriangle \mathsf{SetOf}\,\mathsf{Set}$ and $\bigcup : \mathsf{SetOf}\,\mathsf{Set} \rightarrowtriangle \mathsf{Set}$, we can deduce that if f is a function (i.e., $f : \mathsf{Fun}$), then $T = \bigcup(\mathcal{P}(\mathsf{dom}\, f))$ is a set (i.e., $T : \mathsf{Set}$) and is not "undefined" (i.e., $T \neq \bullet$). Our soft types are similar in spirit to the soft types of Mizar [15], but differ in many details.

We define specialized soft type constructors for our features, e.g., using the SetOf soft type constructor we can build the soft type $\mathsf{SetOf}\,\mathsf{Set}$ of those sets that contain only sets. For example, $\{0\} : \mathsf{SetOf}\,\mathsf{Ord}$, because $\{0\}$ is a set containing only ordinals, and $\{\{0\}\} : \mathsf{SetOf}\,\mathsf{Set}$, and also $\{\{0\}\} : \mathsf{SetOf}\,(\mathsf{SetOf}\,\mathsf{Ord})$.

The feature **GZF** provides *generalized Zermelo/Fraenkel* sets. $\mathsf{GZF}_{\mathsf{defs}}$ defines the cargo soft type SetMem, a soft type for objects that belong to some set in the domain. **Ordinal** provides ordinal numbers. **OrdRec** provides an operator for recursion on ordinals, which is crucial for building models of GSTs. Adding the **Ordinal** and **OrdRec** features saved us at least a month of development time because it allows us to pass the development of ordinal recursion from a type implementing `ZFC_in_HOL` to a GST whose model is built within that type. **Function** provides functions with a function application relation app, operators to find the domain and range of a function, a partial function space operator $\nrightarrow$, and a function-building operator mkFun. **Exc** provides the *exception object* $\bullet$ (spoken "boom"). The important behavior of $\bullet$ is given by axioms generated when combining features that use cargo soft types (e.g., SetMem, FunMem) to ensure that $\bullet$ can not occur within container objects (e.g., sets, functions).

Our current design requires each object in a GST's domain to be "owned" by exactly one feature. However, classes associated with a feature may be used independently of that feature and can be instantiated by any GST's domain provided some collections of objects in the domain can be found that satisfy the requirements the class places on parts of the type. For example, $\mathsf{Function}$ can be

[5] In all of our example features below, we use the symbol $\circ$ decorated with a subscript as the name of the default parameter and we place the default parameter last in the parameter list, but this is just a convention.

$$\begin{aligned}
\mathsf{typList}_i(P \twoheadrightarrow Q) &:= b_i : P \,\#\, \mathsf{typList}_{i+1}(Q) \\
\mathsf{typList}_i(\Pi\, P\, Q) &:= b_i : P \,\#\, \mathsf{typList}_{i+1}(Q\, b_i) \\
\mathsf{typList}_i(R) &:= \diamond \\
\mathsf{otherwise}(\kappa, [b_1 : P_1, \ldots, b_n : P_n], D) &:= \\
\forall\, b_1, \ldots, b_n.\, (\neg\, b_1 : P_1 \vee \ldots \vee \neg\, b_n : P_n) &\to (\kappa\, b_1 \ldots b_n = D) \\
\mathsf{allOtherwise}(\mathcal{F}, D) := [\, \mathsf{otherwise}(\kappa, \mathsf{typList}(R), D) &\mid (\kappa : R)\ \epsilon\colon \mathsf{axioms}(\mathcal{F}), \\
& R \equiv (P \twoheadrightarrow Q) \text{ or } R \equiv (\Pi\, x : P.\, Q)\,] \\
\hline
\mathsf{cover}([P_1, \ldots, P_n]) &:= (P_1 \sqcup \ldots \sqcup P_n = \top) \\
\mathsf{disjoint}([P_1, \ldots, P_n]) &:= [\, (P_1 \sqcap P_2 = \bot), \ldots, (P_1 \sqcap P_n = \bot), \ldots, (P_{n-1} \sqcap P_n = \bot)\,] \\
\hline
\mathsf{admitCargo}(P, [\, Q_1, \ldots, Q_n\,]) &:= (Q_1 \sqcup \ldots \sqcup Q_n \sqsubseteq P) \\
\mathsf{restrictCargo}(P, [\, Q_1, \ldots, Q_n\,]) &:= ((Q_1 \sqcup \ldots \sqcup Q_n) \sqcap P = \bot) \\
\mathsf{cargoAx}(\mathcal{F}, \mathcal{W}, \mathcal{B}) &:= [\, \mathsf{admitCargo}(\mathsf{cargo}(\mathcal{F}), [\, \mathsf{logo}(\mathcal{G}) \mid \mathcal{G}\ \epsilon\colon \mathcal{W}, \mathcal{G}\ \not\epsilon\colon \mathcal{B}\,]), \\
& \qquad \mathsf{restrictCargo}(\mathsf{cargo}(\mathcal{F}), [\, \mathsf{logo}(\mathcal{G}) \mid \mathcal{G}\ \epsilon\colon \mathcal{B}\,])\,]
\end{aligned}$$

Fig. 6. 'Otherwise' axioms for operators, logo axioms, and cargo axioms.

instantiated by sets of ordered pairs (cf. `GZF/SetRel.thy`), and we do this to build a GST at type $\mathbf{V}$ (our founder domain).

3.3 Feature Combination. A *feature configuration* is a triple $[\mathcal{F}, D, \mathcal{B}]$, where D is a term to be identified with $\mathsf{default}(\mathcal{F})$, and $\mathcal{B}$ is a *blacklist* of features whose objects will be excluded from the internal structure of objects of feature $\mathcal{F}$. GSTs are classes defined by combining feature configurations. Figure 6 defines operations that generate extra axioms.

Most of our features use soft typing axioms constraining operator behavior, e.g., $\mathsf{dom} : \mathsf{Fun} \twoheadrightarrow \mathsf{Set}$ specifies that dom yields a set when applied to a function. These soft typing judgements have a special status: they are also used to generate axioms specifying what operators do in other cases, e.g., specifying that $\mathsf{dom}\, B$ will yield $\bullet$ when B is not a function. The formula $\mathsf{otherwise}(\kappa, [\, b_1 : P_1, \ldots, b_n : P_n\,], D)$ produces an axiom that identifies D with the result of applying κ to arguments that do not satisfy at least one of the $P_1, \ldots, P_n$. The formula list $\mathsf{allOtherwise}(\mathcal{F}, D)$ calls $\mathsf{otherwise}$ on each parameter κ with an axiom in $\mathcal{F}$ of the form $\kappa : P \twoheadrightarrow Q$ or $\kappa : (\Pi\, x : P.\, Q)$. Hence, $\mathsf{allOtherwise}(\mathbf{Function}, \bullet)$ generates the axiom $\forall\, b.\, \neg\, b : \mathsf{Fun} \to \mathsf{dom}\, b = \bullet$, among others. We expect some users might prefer other policies and we aim to make this more flexible.

The formula $\mathsf{cover}([\, P_1, \ldots, P_n\,])$ ensures that $B : P_i$ holds for some P_i, and $\mathsf{disjoint}([\, P_1, \ldots, P_n\,])$ ensures that both $B : P_i$ and $B : P_j$ cannot hold for $i \neq j$. The formula $\mathsf{admitCargo}(P, [\, Q_1, \ldots, Q_n\,])$ states that $Q_1, \ldots, Q_n$ are all subtypes of P, and $\mathsf{restrictCargo}(P, [\, Q_1, \ldots, Q_m\,])$ states that if $B : Q_i$ for any $i \leq m$, then $\neg\, B : P$. The formulas of $\mathsf{cargoAx}$ constrain the internal structure of the objects of a feature, e.g., $\mathsf{cargoAx}(\mathbf{GZF}, [\, \mathbf{GZF}, \mathbf{Exc}\,], [\, \mathbf{Exc}\,])$ stands for: $[\, (\mathsf{Set} \sqsubseteq \mathsf{SetMem}), (\mathsf{Exc} \sqcap \mathsf{SetMem} = \bot)\,]$.

Figure 7 defines mkGST and gives an example GST named ZF^+. The operation $\mathsf{mkGST}(\mathsf{spec})$ defines a class $\mathcal{C}$ from a list spec of feature configurations. The

Given $\mathsf{spec} \equiv [[\mathcal{F}_1, D_1, \mathcal{B}_1], \ldots, [\mathcal{F}_n, D_n, \mathcal{B}_n]]$, then:

$$\begin{aligned}
\mathsf{GST}_{\mathsf{axioms}}(\mathsf{spec}) &:= \mathsf{allOtherwise}(\mathcal{F}_1, D_1) \mathbin{@} \ldots \mathbin{@} \mathsf{allOtherwise}(\mathcal{F}_n, D_n) \\
&\quad \mathbin{@} \mathsf{disjoint}\,[\,\mathsf{logo}(\mathcal{F}_1), \ldots, \mathsf{logo}(\mathcal{F}_n)\,] \\
&\quad \mathbin{@} [\,\mathsf{cover}\,[\,\mathsf{logo}(\mathcal{F}_1), \ldots, \mathsf{logo}(\mathcal{F}_n)\,]\,] \\
&\quad \mathbin{@} \mathsf{cargoAx}(\mathcal{F}_1, [\,\mathcal{F}_1, \ldots, \mathcal{F}_n\,], \mathcal{B}_1) \\
&\quad \mathbin{@} \ldots \mathbin{@} \mathsf{cargoAx}(\mathcal{F}_n, [\,\mathcal{F}_1, \ldots, \mathcal{F}_n\,], \mathcal{B}_n) \\
\mathsf{GST}_{\mathsf{defs}}(\mathsf{spec}) &:= [\,\mathsf{default}(\mathcal{F}_1) = D_1, \ldots, \mathsf{default}(\mathcal{F}_n) = D_n\,] \\
\mathsf{mkGST}(\mathsf{spec}) &:= [\,[\,\mathsf{class}(\mathcal{F}_1), \ldots, \mathsf{class}(\mathcal{F}_n)\,], \diamond, \mathsf{GST}_{\mathsf{axioms}}(\mathsf{spec}), \mathsf{GST}_{\mathsf{defs}}(\mathsf{spec})\,]
\end{aligned}$$

$$\mathsf{ZF}^+_{\mathsf{spec}} := [\,[\,\mathbf{GZF}, \bullet, [\,\mathbf{Exc}\,]\,], [\,\mathbf{Ordinal}, \bullet, \diamond], [\,\mathbf{Function}, \bullet, [\,\mathbf{Exc}\,]\,], [\,\mathbf{Exc}, \bullet, \diamond]\,]$$

$$\mathsf{ZF}^+ := \mathsf{mkGST}(\mathsf{ZF}^+_{\mathsf{spec}})$$

Fig. 7. Generating GSTs from specifications and ZF^+, an example GST.

dependencies of $\mathcal{C}$ are the classes of the features in spec, the axioms of $\mathcal{C}$ are given by $\mathsf{GST}_{\mathsf{axioms}}(\mathsf{spec})$, and the definitions of $\mathcal{C}$ are a list of simple definitions of the default parameter of each feature in spec given by $\mathsf{GST}_{\mathsf{defs}}(\mathsf{spec})$. Our example GST ZF^+ has sets, non-set ordinals, non-set functions, and a distinguished non-set $\bullet$. Each feature is configured to use $\bullet$ as its default value, and $\bullet$ is blacklisted from the internal structure of sets, and functions.

4 Examples of Working in a GST

These examples assume parameter instantiations of the class ZF^+ and all the classes in its dependencies (i.e., GZF, Ordinal, Function, and Exception) at the type d_0. This means that from the class ZF^+ and each class in its dependencies deductions can use definitions of the constants in the class parameters as well as the class definitions and the class axioms. When we say "define X", we mean "add the indicated simple definition for X to the definitions used in deductions".

Ordinal Left-subtraction. Suppose we already have an infix ordinal addition operator such that if $i, j : \mathsf{Ord}$ then $i + j : \mathsf{Ord}$, and otherwise $i + j = \bullet$. Define the *left-subtraction* operator $-_{\mathsf{left}} :: \mathsf{d}_0 \Rightarrow \mathsf{d}_0 \Rightarrow \mathsf{d}_0$ on ordinals:

$$-_{\mathsf{left}} = (\lambda\, i\, j.\, \iota\, k.\, i = j + k \,\mathsf{else}\, \bullet)$$

Given ordinals $i, j : \mathsf{Ord}$ where $j < i$ or $j = i$, the term $i -_{\mathsf{left}} j$ is the unique ordinal k such that $i = j + k$. If $i < j$ or if either i or j are not ordinals, then $i -_{\mathsf{left}} j = \bullet$. For example:

$$5 -_{\mathsf{left}} 3 = (\iota\, k.\, 5 = 3 + k \,\mathsf{else}\, \bullet) = 2$$

$$3 -_{\mathsf{left}} 5 = (\iota\, k.\, 3 = 5 + k \,\mathsf{else}\, \bullet) = \bullet$$

$$3 -_{\mathsf{left}} \emptyset = (\iota\, k.\, 3 = \emptyset + k \,\mathsf{else}\, \bullet) = \bullet$$

Function Application and Some Other Function Operators. Define the *function application* infix operator $(`) :: \mathsf{d}_0 \Rightarrow \mathsf{d}_0 \Rightarrow \mathsf{d}_0$ such that if $f : \mathsf{Fun}$ and

$x \in \mathsf{dom}\, f$, then $f \mathbin{`} x$ is the value of $f$ at $x$, and otherwise $f \mathbin{`} x = \bullet$:

$$(`) = (\lambda\, f\, x.\, \imath\, y.\, \mathsf{app}\, f\, x\, y\, \mathsf{else}\, \bullet)$$

Define an operator $(\dot{\lambda})$ that when given a domain $x : \mathsf{Set}$ extracts from an operator $t :: \mathsf{d}_0 \Rightarrow \mathsf{d}_0$ a function of type d_0:

$$\dot{\lambda} = (\lambda\, x\, t.\, \mathsf{mkFun}\, x\, (\lambda\, b\, c.\, b \in x \wedge c = t\, b))$$

We furthermore adopt the following nice notation: $(\dot{\lambda}\, b \in x.\, B) := (\dot{\lambda}\, x\, (\lambda\, b.\, B))$. Define an operator to lift a binary operator on objects of type d_0 to a binary operator on functions:

$$\mathsf{lift} = \lambda\, o\, f\, g.\, (\dot{\lambda}\, b \in \mathsf{dom}\, f \cap \mathsf{dom}\, g.\, o\, (f \mathbin{`} b)\, (g \mathbin{`} b))$$

Define an operator $\mathsf{FunRet} :: (\mathsf{d}_0 \Rightarrow \star) \Rightarrow \mathsf{d}_0 \Rightarrow \star$ that takes a soft type P and yields the soft type of functions whose return values always satisfy P:

$$\mathsf{FunRet} = (\lambda\, p\, f.\, f : \mathsf{Fun} \wedge \mathsf{ran}\, f : \mathsf{SetOf}\, p)$$

Pointwise Left-subtraction on Ordinal-valued Functions. A function $f : \mathsf{Fun}$ is *ordinal-valued* iff $f : \mathsf{FunRet\,Ord}$. Thus, $\mathsf{lift}\, (-_{\mathsf{left}})\, f\, g$ computes the pointwise left-subtraction of ordinal-valued functions $f, g : \mathsf{FunRet\,Ord}$. For example, consider functions f and g with $\mathsf{dom}\, f$ and $\mathsf{dom}\, g$ being $\mathsf{predSet}\, \omega$ (the set of all ordinals less than ω), where $f \mathbin{`} j = j + 2$ and $g \mathbin{`} j = j + 1$ for all $j \in \mathsf{predSet}\, \omega$. Clearly f, g are ordinal-valued. Hence for any ordinal $j < \omega$,

$$\begin{aligned}(\mathsf{lift}\, (-_{\mathsf{left}})\, f\, g) \mathbin{`} j &= (\dot{\lambda}\, b \in \mathsf{dom}\, f \cap \mathsf{dom}\, g.\, (f \mathbin{`} b) -_{\mathsf{left}} (g \mathbin{`} b)) \mathbin{`} j \\ &= (f \mathbin{`} j -_{\mathsf{left}} g \mathbin{`} j) = (j + 2) -_{\mathsf{left}} (j + 1) \\ &= 2 -_{\mathsf{left}} 1 = 1\end{aligned}$$

Combining Operators. Define an operator $\mathsf{override}$ that takes two binary operators $o_1, o_2 :: \mathsf{d}_0 \Rightarrow \mathsf{d}_0 \Rightarrow \mathsf{d}_0$ and two predicates $t_1, t_2 :: \mathsf{d}_0 \Rightarrow \mathsf{d}_0 \Rightarrow \star$ saying when to use each operator and builds a new operator by combining them:

$$\begin{aligned}\mathsf{override}\, t_1\, o_1\, t_2\, o_2 = (\lambda\, x\, y.\, &\mathsf{if}\ (t_1\, x\, y)\ \mathsf{then}\ (o_1\, x\, y) \\ &\mathsf{else\ if}\ (t_2\, x\, y)\ \mathsf{then}\ (o_2\, x\, y) \\ &\mathsf{else}\ \bullet)\end{aligned}$$

Define an infix operator $(-)$ that combines ordinal left-subtraction with ordinal left-subtraction lifted to ordinal-valued functions:

$$(-) = \mathsf{override}\, (\lambda\, x\, y.\ \ x, y : \mathsf{Ord})\, (-_{\mathsf{left}})\, (\lambda\, x\, y.\ \ x, y : \mathsf{FunRet\,Ord})\, (\mathsf{lift}\, (-_{\mathsf{left}}))$$

If i, j are ordinals, then $i - j = i -_{\mathsf{left}} j$. If f, g are ordinal-valued functions, then $f - g$ is the function such that if $j \in \mathsf{dom}\, f$ and $j \in \mathsf{dom}\, g$, then $(f - g) \mathbin{`} j = (f \mathbin{`} j) -_{\mathsf{left}} (g \mathbin{`} j)$, and otherwise $(f - g) \mathbin{`} j = \bullet$. Because $\mathsf{Ord} \sqcap \mathsf{Fun} = \bot$ is an axiom of ZF^+, we know that at most one of $x, y : \mathsf{Ord}$ and $x, y : \mathsf{FunRet\,Ord}$ can be true, so there is no possibility of $(-)$ using $(-_{\mathsf{left}})$ where the user might have intended instead for $(-)$ to use $(\mathsf{lift}\, (-_{\mathsf{left}}))$.

5 Building a Model for ZF$^+$ in ZFC

Normally, using our example GST ZF$^+$ is done in a type that it is instantiated at. Such a type could be gained by including the axioms of ZF$^+$ in the hypotheses of deductions (i.e., $\mathsf{ZF}^+_{\mathsf{axioms}}[\![\mathsf{d}_0]\!] \subseteq \Gamma$). For confidence in ZF$^+$'s consistency, we opt to *define* d_0 in terms of a model built in $\mathbf{V}$, a type axiomatized by Paulson's "ZFC in HOL".[6] Models of GSTs are cumulative hierarchies of sets defined via ordinal recursion. We build models of GSTs within other GSTs using classes called *model components* that correspond to features. We use Isabelle/ML code to generate terms, theorems, and proof states that assist in instantiating these classes and using them as interpretations of GST features. Details of our approach may be seen in the appendix of the longer version of this paper.

The only axioms (i.e., excluding simple definitions) used for this are the members of the sets HOL and ZFCinHOL. From this point, only *definitional mechanisms* (as defined by Kunčar and Popescu [11]) are used to define ZF$^+$ in d_0 (i.e., class instantiation, `typedef`, `lift_definition`). Hence, if HOL and ZFCinHOL are consistent, then so are the axioms of ZF$^+$.

5.1 Building a model in V. To build a model in $\mathbf{V}$, our method needs the GZF, Ordinal, OrdRec, and Function classes to be instantiated at $\mathbf{V}$.

The axioms in HOL and ZFCinHOL provide ZF set theory which supports von Neumann ordinals and transfinite recursion. Hence instantiations for GZF, Ordinal, OrdRec are achieved by choosing correct definitions for each parameter, and then using automatic proof methods to discharge proof obligations. For Function, we have a generic result that this class may be instantiated in any GST with the GZF feature, using sets of Kuratowski ordered pairs.

We define a model in $\mathbf{V}$ and a soft type $\mathbb{M} :: \mathbf{V} \Rightarrow \star$ of all objects in the model. We then do an Isabelle/HOL type definition to define d_0 as a type isomorphic to the model of ZF$^+$ built in $\mathbf{V}$ (i.e., `typedef` $\mathsf{d}_0 = \mathsf{Collect}\,\mathbb{M}$).[7]

5.2 Instantiating ZF$^+$ in d_0. To instantiate the ZF$^+$ class at d_0, we must provide instantiations for each parameter of ZF$^+$, and prove each axiom. This is achieved with help from the Lifting and Transfer [7] packages. The parameters of each feature of ZF$^+$ — namely GZF, Ordinal, Function and Exc — are instantiated using Isabelle/ML code calling the Lifting package to lift constants acting on the model in $\mathbf{V}$ to d_0. To prove the axioms of each feature, the Transfer package is used to transform each subgoal into an equivalent statement about the model. Our approach only needs the transferred versions of a feature's axioms to be proved once — the proofs may be reused when building a model of any GST using that feature. The axioms of ZF$^+$ generated in the feature combination process (e.g., cover, disjoint) must also be proved on d_0. This is also achieved with Transfer, but these statements are proved manually.

[6] Interested readers can see the definition of the ZF$^+$ class in `GST_Features.thy`, and the type definition of d_0 and instantiation of the ZF$^+$ class in `Founder/Test.thy`.

[7] The operator Collect converts a predicate to a HOL-set as required by `typedef`.

6 Related and Future Work

Other Related Work. Two bodies of work are notable in being similar in spirit to the way we assemble GSTs, specifically work by Farmer et al. on "little theories" [6], which emphasizes small theories that are then connected together by morphisms, and work led by Rabe on the MMT framework [14], which emphasizes combining even smaller theory bits with morphisms.

Future Work. We aim to further develop generalized set theories and our Isabelle/HOL formalization in the following directions. *Importing features from Isabelle/HOL.* The "ZFC in HOL" development that we bootstrap from provides support (packaged up in two type classes) for obtaining structures within the set theory with the same behavior as other Isabelle/HOL types. We aim to be able to pass such structures on to GSTs we build in the form of extra features. *Foundation.* We have not yet implemented the axiom of foundation, which states that there are no infinite chains going by steps from an object to any of its "children". Many use cases do not need this, so it is not crucial, but some of our future work will depend on it. We plan to handle this as described in previous work on formalizing GSTs [5]. *Overloading.* Isabelle supports overloading (via classes or *ad hoc* commands) so symbols can have different meanings at different types. However, we want to give symbols different meanings for different features within one type, a GST's domain. For example, we want machinery that will support overloading $\in$ to work both with Set objects and groups within a GST. We also want to allow symbols for set operations like $\in$ to be used both with soft types (which are close to Isabelle/HOL sets) and Set objects in a GST. *Universes.* To support some applications in category theory and also for building models of complicated type theories to embed them within a GST, we aim to add a feature that adds some kind of universe axiom like in TG, so those who need it could have a much larger and more powerful GST. *Automation.* Others are working on automation of finding and using soft types and we aim to build on any progress they make. Our model-building process currently needs lots of manual proving, but much can be automated and/or given a nicer interface.

References

1. P. Aczel. Generalised set theory. In *Logic, Language and Computation, Volume 1*. CSLI Publications, 1996.
2. P. Aczel, R. Lunnon. Universes and parameters. In *Situation Theory and Its Applications, Volume 2*. CSLI Publications, 1991.
3. C. E. Brown, C. Kaliszyk, K. Pąk. Higher-order Tarski Grothendieck as a foundation for formal proof. In *10th Int'l Conf. Interactive Theorem Proving (ITP 2019)*. Dagstuhl Publishing, 2019.
4. C. Dunne, J. B. Wells, F. Kamareddine. Adding an abstraction barrier to ZF set theory. In *Intelligent Computer Mathematics*, vol. 12236 of *LNCS*. Springer, 2020.
5. C. Dunne, J. B. Wells, F. Kamareddine. Generating custom set theories with non-set structured objects. In *Intelligent Computer Mathematics: 14th Int'l Conf., CICM 2021, Proc.*, vol. 12833 of *LNCS*. Springer, 2021.

6. W. M. Farmer, J. D. Guttman, F. J. Thayer. Little theories. In *Automated Deduction: CADE-11*, vol. 607 of *LNCS*. Springer-Verlag, 1992.
7. B. Huffman, O. Kunčar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In *Certified Programs and Proofs*, vol. 8307 of *LNCS*. Springer, 2013.
8. C. Kaliszyk, K. Pąk. Semantics of Mizar as an Isabelle object logic. *J. Automated Reasoning*, 63, 2019.
9. K. Kappelmann, J. Chen, A. Krauss. Isabelle/Set, 2022. Git repository on github.com. Committers include also Cezary Kaliszyk and Karol Pąk.
10. A. Krauss. Adding soft types to Isabelle, 2010. Unpublished paper on author's web page.
11. O. Kunčar, A. Popescu. Safety and conservativity of definitions in HOL and Isabelle/HOL. *Proceedings of the ACM on Programming Languages*, 2, 2017.
12. O. Kunčar, A. Popescu. A consistent foundation for Isabelle/HOL. *Journal of Automated Reasoning*, 62(4), 2019.
13. L. C. Paulson. Zermelo Fraenkel set theory in higher-order logic. *Archive of Formal Proofs*, 2019. Formal proof development available at `https://isa-afp.org/entries/ZFC_in_HOL.html`.
14. F. Rabe. The future of logic: Foundation-independence. *Logica Universalis*, 10, 2015.
15. F. Wiedijk. Mizar's soft type system. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 2007.