Noname manuscript No.
(will be inserted by the editor)

From Types to Sets by Local Type Definition in Higher-Order Logic

Ondřej Kunčar · **Andrei Popescu**

the date of receipt and acceptance should be inserted later

Abstract Types in Higher-Order Logic (HOL) are naturally interpreted as nonempty sets. This intuition is reflected in the type definition rule for the HOL-based systems (including Isabelle/HOL), where a new type can be defined whenever a nonempty set is exhibited. However, in HOL this definition mechanism cannot be applied *inside proof contexts*. We propose a more expressive type definition rule that addresses the limitation and we prove its consistency. This higher expressive power opens the opportunity for a HOL tool that relativizes type-based statements to more flexible set-based variants in a principled way. We also address particularities of Isabelle/HOL and show how to perform the relativization in the presence of type classes.

1 Motivation

The proof assistant community is mainly divided in two successful camps. One camp, represented by provers such as Agda [10], Coq [9], Matita [6] and Nuprl [13], uses expressive type theories as a foundation. The other camp, represented by the HOL family of provers (including HOL4 [2], HOL Light [18], HOL Zero [4] and Isabelle/HOL [34]), remains faithful to a form of classic set theory typed using simple types with rank-1 polymorphism. (Other successful provers, such as ACL2 [25] and Mizar [8], could be seen as being closer to the HOL camp, although technically they are not based on HOL.)

According to the HOL school of thought, a main goal is to acquire a sweet spot: keep the logic as simple as possible while obtaining *sufficient expressiveness*. The notion of sufficient expressiveness is of course debatable, and has been debated. For example, PVS [38] includes dependent types (but excludes polymorphism), HOL-Omega [22] adds first-class

This is the extended, journal version of the conference paper [27], submitted to the JAR special issue dedicated to ITP 2016.

Ondřej Kunčar
Fakultät für Informatik, Technische Universität München, Germany
E-mail: kuncar@in.tum.de

Andrei Popescu
Department of Computer Science, School of Science and Technology, Middlesex University, UK
E-mail: a.popescu@mdx.ac.uk

type constructors to HOL, and Isabelle/HOL adds ad hoc overloading of polymorphic constants. In this paper, we want to propose a gentler extension of HOL. We do not want to promote new "first-class citizens," but merely to give better credit to an old and venerable HOL citizen: the notion of types emerging from sets.

The problem that we address in this paper is best illustrated by an example. Let $\mathsf{lists} : \alpha\ \mathsf{set} \to \alpha\ \mathsf{list}\ \mathsf{set}$ be the constant that takes a set A and returns the set of lists whose elements are in A, and $\mathsf{P} : \alpha\ \mathsf{list} \to \mathsf{bool}$ be another constant (whose definition is not important here). Consider the following two statements, which use either types or sets to represent semantically the same mathematical fact (for clarity, we extend the usual HOL syntax by explicitly quantifying over types at the outermost level as in HOL-Omega):

$$\forall\alpha.\ \exists xs_{\alpha\ \mathsf{list}}.\ \mathsf{P}\ xs \tag{1}$$

$$\forall\alpha.\ \forall A_{\alpha\ \mathsf{set}}.\ A \neq \emptyset \longrightarrow (\exists xs \in \mathsf{lists}\ A.\ \mathsf{P}\ xs) \tag{2}$$

The formula (2) is a relativized form of (1), quantifying not only over all types α, but also over all their nonempty subsets A, and correspondingly relativizing the quantification over all lists to quantification over the lists built from elements of A. We call theorems such as (1) *type based* and theorems such as (2) *set based*.

Type-based theorems have obvious advantages compared to the set-based ones. First, they are more concise. Moreover, automatic proof procedures work better for them, thanks to the fact that they encode properties more rigidly and more implicitly, namely, in the HOL types (such as membership to α list) and not via formulas (such as membership to the set $\mathsf{lists}\ A$). On the downside, type-based theorems are less flexible, and therefore unsuitable for some developments. Indeed, when working with mathematical structures, it is often the case that they have the desired property only on a proper subset of the whole type. For example, a function f from τ to σ may be injective or continuous only on a subset of τ.

When wishing to apply type-based theorems from the library to deal with such situations, users are forced to produce ad hoc workarounds: they can often define a new (global) type isomorphic to the concrete set, which HOL allows only if the set can be represented by a closed term.[1] But quite often the set depends on some local variables or local definitions (typically because it was constructed inside of a proof), in which case one needs the set-based relativization of the type-based library theorem. In the most striking cases, such a relativization is created manually. For example, in Isabelle/HOL there exists the constant $\mathsf{inj\text{-}on}\ A\ f = (\forall x\ y \in A.\ f\ x = f\ y \longrightarrow x = y)$ together with a small library about functions being injective only on a subset of a type. In summary, while it is easier to reason about type-based statements such as (1), the set-based statements such as (2) are more general and more widely applicable.

An additional nuance to this situation is specific to Isabelle/HOL, which allows users to annotate types with Haskell-like [41] type-class constraints [35]. This provides a further level of implicit reasoning. For example, instead of explicitly quantifying a statement over an associative operation $*$ on a type σ, one marks σ as having class $\mathsf{semigroup}$ (which carries implicitly the assumptions). This would also need to be reversed when relativizing from types to sets. If (1) made the assumption that α is a semigroup, as in $\forall\alpha_{\mathsf{semigroup}}.\ \exists xs_{\alpha\ \mathsf{list}}.\ \mathsf{P}\ xs$, then (2) would need to quantify universally not only over A, but also over a binary operation on A, and explicitly assume it to be associative.

Isabelle provides a large collection of arithmetic simplification procedures, which serve as an example of an automation that works better for type-based goals than for the set-based ones

[1] Let us recall that HOL does not allow for dependent types.

since they work smoothly only if the corresponding simplification rules apply to the whole type. For example, the cancellation procedure can discharge the goal $x - x = 0$ only if x's type is restricted to an appropriate algebraic structure such as α_{group}, in which case the goal can be easily rewritten to True. In the set-based setting, in which x has an unrestricted type α, the formula does not hold in general. The cancellation would work only if the simplifier could prove that $x \in G$ and group G for some G, which would have to be synthesized or obtained from the proof context. Moreover, since G itself could be an arbitrary term, the simplifier could rewrite G in such a way that $x \in G$ could not be proved anymore.

The aforementioned problem, of the mismatch between type-based theorems from libraries and set-based versions needed by users, shows up regularly in requests posted on the Isabelle community mailing lists. Here is an example [43]: *Various lemmas* [from the theory Finite_Set] *require me to show that f* [commutes with ∘] *for all x and y. This is a too strong requirement. I can show that it holds for all x and y in A, but not for all x and y in general.*

Often, users feel the need to convert entire libraries from type-based theorems to set-based ones. For example, our colleague Fabian Immler writes about his large formalization experience [24, §5.7]: *The main reason why we had to introduce this new type* [of finite maps] *is that almost all topological properties are formalized in terms of type classes, i.e., all assumptions have to hold on the whole type universe. It feels like a cleaner approach* [would be] *to relax all necessary topological definitions and results from types to sets because other applications might profit from that, too.*

A prophylactic alternative is of course to develop the libraries in a set-based fashion from the beginning, agreeing to pay the price in terms of verbosity and lack of automation. And numerous developments in different HOL-based provers do just that [5, 11, 12, 21, 31].

In this paper, we propose an alternative that gets the best of both worlds: *Prove easily and still be flexible.* More precisely, develop the libraries type based, but export the results set based. We start from the observation that, from a set-theoretic semantics standpoint, the theorems (1) and (2) are equivalent: they both state that, for every nonempty collection of elements, there exists a list of elements from that collection for which P holds. Unfortunately, the HOL logic in its current form is blind to one direction of this equivalence: assuming that (1) is a theorem, one cannot prove (2). Indeed, in a proof attempt of (2), one would fix a nonempty set A and, to invoke (1), one would need to define a new type corresponding to A—an action not currently allowed inside a HOL proof context.

In this paper, we propose a natural and sound extension to HOL (and to Isabelle/HOL) to enable type definitions to be emulated inside HOL proof contexts, which allows us to prove the above-mentioned equivalences. We will also show how this can be used to leverage user experience as outlined above.

The paper is organized as follows. In Section 2, we recall the logics of HOL and Isabelle/HOL. In Section 3, we describe the envisioned extension of HOL: adding a new rule for simulating type definitions in proof contexts. In Section 4, we prove that the new rule is a consistent addition to HOL. In Section 5, we demonstrate how the new rule allows us to relativize type-based theorems to set-based ones in HOL. Due to the presence of type classes, we need to extend Isabelle/HOL's logic further to achieve the relativization—this is the topic of Section 6. In Section 7, we outline the process of performing the relativization in a principled and automated way. In Section 8, we describe an application of our results to a more general version of relativization, namely from types to terms. Finally, in Section 9 we describe limitations of our approach and outline our future work.

Starting from Isabelle2016-1, the implementation of the proposed logical extensions and some examples discussed in this paper are part of the Isabelle distribution [3].

2 HOL and Isabelle/HOL Recalled

In this section, we briefly recall the logics of HOL and Isabelle/HOL. We distinguish between the *core logic*, which is common to HOL and Isabelle/HOL, and the *definitional mechanisms*, which differ between HOL and Isabelle/HOL.

2.1 Core Logic

The core logic of HOL and Isabelle/HOL is classical Higher-Order Logic with rank-1 polymorphism, Hilbert choice and the Infinity axioms.

We chose one of the equivalent formulations of the core logic of HOL in this paper. Other authors [15, 17, 33] might start from different primitives or use slightly different set of primitive inference rules (or add derived rules as primitives for performance reasons in a concrete implementation). All these approaches are equivalent and do not affect our results.

We fix the following:

- an infinite set TVar, of *type variables*, ranged by α, β
- an infinite set VarN, of *(term) variable names*, ranged by x, y, z
- a set K of symbols, ranged by κ, called *type constructors*, containing three special symbols: "bool", "ind" and "$\to$" (aimed at representing the type of booleans, an infinite type and the function type constructor, respectively)

We fix a function $\mathsf{arOf} : K \to \mathbb{N}$ giving arities to type constructors, such that $\mathsf{arOf}(\mathsf{bool}) = \mathsf{arOf}(\mathsf{ind}) = 0$ and $\mathsf{arOf}(\to) = 2$. If $\mathsf{arOf}(\kappa) = n$, we say that κ is an n-ary type constructor. Types, ranged by σ, τ, are defined as follows:

$$\sigma = \alpha \mid (\sigma_1, \ldots, \sigma_{\mathsf{arOf}(\kappa)})\, \kappa$$

Thus, a type is either a type variable or an n-ary type constructor κ postfix-applied to a number of types corresponding to its arity. If $n = 1$, instead of $(\sigma)\,\kappa$ we write $\sigma\,\kappa$.

Finally, we fix the following:

- a set Const, ranged over by c, of symbols called *constants*, containing five special symbols: "$\longrightarrow$", "$=$", "ε", "zero" and "suc" (aimed at representing logical implication, equality, Hilbert choice of some element from a type, zero and successor, respectively)
- a function $\mathsf{tpOf} : \mathsf{Const} \to \mathsf{Type}$ associating a type to every constant, such that:

$$\begin{array}{ll} \mathsf{tpOf}(\longrightarrow) = \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool} & \mathsf{tpOf}(\mathsf{zero}) = \mathsf{ind} \\ \mathsf{tpOf}(=) = \alpha \to \alpha \to \mathsf{bool} & \mathsf{tpOf}(\mathsf{suc}) = \mathsf{ind} \to \mathsf{ind} \\ \mathsf{tpOf}(\varepsilon) = (\alpha \to \mathsf{bool}) \to \alpha & \end{array}$$

$\mathsf{TV}(\sigma)$ is the set of variables of a type σ. Given a function $\rho : \mathsf{TVar} \to \mathsf{Type}$, its *support* is the set of type variables where ρ is not the identity: $\mathsf{supp}(\rho) = \{\alpha \mid \rho(\alpha) \neq \alpha\}$. A *type substitution* is a function $\rho : \mathsf{TVar} \to \mathsf{Type}$ with finite support. We let TSubst denote the set of type substitutions. Each $\rho \in \mathsf{TSubst}$ extends to a function $\rho : \mathsf{Type} \to \mathsf{Type}$ by defining $\rho(\alpha) = \rho(\alpha)$ and $\rho((\sigma_1, \ldots, \sigma_n)\,\kappa) = (\rho(\sigma_1), \ldots, \rho(\sigma_n))\,\kappa$.

We say that σ is an *instance* of τ *via a substitution* of $\rho \in \mathsf{TSubst}$, written $\sigma \leq_\rho \tau$, if $\rho(\tau) = \sigma$ and $\mathsf{supp}(\rho) \subseteq \mathsf{TV}(\tau)$. We say that σ is an *instance* of τ, written $\sigma \leq \tau$, if there exists $\rho \in \mathsf{TSubst}$ such that $\sigma \leq_\rho \tau$. Notice that if $\sigma \leq \tau$, there is a unique ρ such that $\sigma \leq_\rho \tau$. We say that σ *is in* τ if σ is syntactically contained in τ.

A *(typed) variable* is a pair of a variable name x and a type σ, written x_σ. Let Var denote the set of all variables. A *constant instance* is a pair of a constant and a type, written c_σ, such that $\sigma \leq \mathsf{tpOf}(c)$. We let CInst denote the set of constant instances.

The tuple $\Sigma = (K, \mathsf{arOf}, \mathsf{Const}, \mathsf{tpOf})$, which will be fixed in what follows, is called a *signature*. This signature's *pre-terms*, ranged over by s, t, are defined by the grammar:

$$t = x_\sigma \mid c_\sigma \mid t_1\ t_2 \mid \lambda x_\sigma.\ t$$

Thus, a pre-term is either a typed variable, or a constant instance, or an application, or an abstraction. As usual, we identify pre-terms modulo alpha-equivalence. Typing is defined as a binary relation between pre-terms and types, written $t : \sigma$, structurally inductively as follows:

$$\frac{x \in \mathsf{VarN}}{x_\sigma : \sigma} \qquad \frac{c \in \mathsf{Const} \qquad \tau \leq \mathsf{tpOf}(c)}{c_\tau : \tau} \qquad \frac{t_1 : \sigma \to \tau \qquad t_2 : \sigma}{t_1\ t_2 : \tau} \qquad \frac{t : \tau}{\lambda x_\sigma.\ t : \sigma \to \tau}$$

A term is a well-typed pre-term if there exists a (necessarily unique) type τ such that $t : \tau$. We let Term be the set of well-typed terms. We can apply a type substitution ρ to a term t, written $\rho(t)$, by applying ρ to the types of all variables and constant instances occurring in t and potentially renaming some bound variables if they would get identified. We say that a constant c (or a type σ) *is in* a term t if t syntactically contains c (or σ respectively). The set $\mathsf{FV}(t)$ is the set of t's free variables. The term t is called *closed* if it has no free variables: $\mathsf{FV}(t) = \emptyset$. When writing terms, we sometimes omit the types of variables if they can be inferred. We sometimes use brackets to indicate that certain types and/or terms are a part of the term, e.g., $t[\alpha]$, $t[\sigma, c_\tau]$.

A *formula* is a term of type bool. The logical constants True and False, formula connectives and quantifiers are defined in the standard way, starting from the implication and equality primitives.

In HOL, types represent "rigid" collections of elements. More flexible collections can be obtained using sets. Essentially, a set on a type σ, also called a subset of σ, is given by a predicate $S : \sigma \to \mathsf{bool}$. Then membership of an element a to S, written $a \in S$, is defined to mean $S\ a$ (which is the same as $S\ a = \mathsf{True}$). HOL systems differ in how abstractly they represent this concept of "sets as predicates". Our paper is agnostic on this in the following sense: we write $\alpha\ \mathsf{set}$ to be either a textual abbreviation for $\alpha \to \mathsf{bool}$ (corresponds to HOL Light, which uses directly the predicate type) or syntactic sugar for predicates (as in HOL4) or an actual type constructor defined to be isomorphic to the predicates (as in Isabelle/HOL). All these approaches yield essentially the same notion and our results apply to all of them.

A proof context Γ and a HOL theory D are finite sets of formulas. We say that (Γ, φ) is a sequent if Γ is a context and φ is a formula. The HOL deduction relation $\vdash$ is a ternary relation between theories, contexts and formulas, $D; \Gamma \vdash \varphi$, and is defined inductively starting from the formulas in D and HOL axioms (containing axioms for equality, infinity, choice, and excluded middle) and applying deduction rules (introduction and elimination of $\longrightarrow$, term and type instantiation, extensionality and β-reduction). Precise definitions of (equivalent variations of) the HOL deduction system can be found elsewhere [17, 28, 36].

The definitional mechanisms for constants and types extend the signature (by adding the newly defined symbol) and the theory (by adding the definitional formula). We call a theory D definitional if D was created by a sequence of theory extensions corresponding either to a constant or a type definition. Since Isabelle/HOL allows for more flexible constant definitions than HOL does, we will cover their definitional mechanisms separately in the next two sections.

A theory D is consistent if it cannot derive False in the empty context, i.e., if $D; \emptyset \vdash \mathsf{False}$ does not hold.

2.2 Definitional Mechanisms of HOL

Most of the systems implementing HOL follow the tradition to discourage their users from using arbitrary underlying theories D and to promote merely *definitional ones*, containing definitions of constants and types.

A *HOL constant definition* is a formula $c_\sigma = t$, where:

- c is a fresh constant of type σ
- t is a term that is closed (i.e., has no free term variables) and whose type variables are included in those of σ

HOL type definitions are more complex entities. They are based on the notion of a newly defined type β being embedded in an existing type α, i.e., being isomorphic to a given nonempty subset S of α via mappings *Abs* and *Rep*. Let ${}_\alpha(\beta \approx S)^{Abs}_{Rep}$ denote the formula expressing this:

$$(\forall x_\beta.\, Rep\ x \in S) \wedge (\forall x_\beta.\, Abs\ (Rep\ x) = x) \wedge (\forall y_\alpha.\, y \in S \longrightarrow Rep\ (Abs\ y) = y)$$

When the user issues a command `typedef` $\tau = S_{\sigma\,\mathsf{set}}$, they are required to discharge the goal $S \neq \emptyset$, after which the system introduces a new type τ and two constants $\mathsf{Abs}^\tau : \sigma \to \tau$ and $\mathsf{Rep}^\tau : \tau \to \sigma$ and adds the axiom ${}_\sigma(\tau \approx S)^{\mathsf{Abs}^\tau}_{\mathsf{Rep}^\tau}$ to the theory.

2.3 Definitional Mechanisms of Isabelle/HOL

While a member of the HOL family, Isabelle/HOL is special w.r.t. constant definitions. Namely, a constant is allowed to be declared with a given type σ and then "overloaded" on various types τ less general than σ and mutually orthogonal. For example, we can have d declared to have type α, and then $\mathsf{d}_{\mathsf{bool}}$ defined to be True and $\mathsf{d}_{\alpha\,\mathsf{list}}$ defined to be $[\mathsf{d}_\alpha]$. We shall write Δ_c for the collection of all types where c has been overloaded. In the above example, $\Delta_{\mathsf{d}} = \{\mathsf{bool}, \alpha\ \mathsf{list}\}$.

The mechanism of overloaded definitions offers broad expressive power. But with power also comes responsibility. The system has to make sure that the defining equations cannot form a cycle. To guarantee that, a binary constant/type dependency relation $\rightsquigarrow$ on types and constants is maintained, where $u \rightsquigarrow v$ holds true iff one of the following holds:

1. u is a constant c that was declared with type σ and v is a type in σ
2. u is a constant c defined as $c = t$ and v is a type or constant in t
3. u is a type σ defined as $\sigma = A$ and v is a type or constant in A

We write $\rightsquigarrow^\downarrow$ for (type-)substitutive closure of the constant/type dependency relation, i.e., if $p \rightsquigarrow q$, the type instances of p and q are in $\rightsquigarrow^\downarrow$. The system accepts only overloaded definitions for which $\rightsquigarrow^\downarrow$ does not contain an infinite chain.

In addition, Isabelle supports user-defined *axiomatic type classes*, which are essentially predicates on types. They effectively improve the type system with the ability to carry implicit assumptions. For example, we can define the type class $\mathsf{finite}(\alpha)$ expressing that α has a finite number of inhabitants. Then, we are allowed to annotate type variables by such predicates, e.g., α_{finite} or $\alpha_{\mathsf{semigroup}}$ from Section 1. Finally, we can substitute a type τ for α_{finite} only if τ has been previously proved to fulfill $\mathsf{finite}(\tau)$.

The axiomatic type classes become truly useful when we use overloaded constants for their definitions. This combination allows the use of Haskell-style type classes. E.g., we can

reason about arbitrary semigroups by declaring a global constant $* : \alpha \to \alpha \to \alpha$ and defining the HOL predicate $\mathsf{semigroup}(\alpha)$ stating that $*$ is associative on α.

In this paper, we are largely concerned with results relevant for the entire HOL family of provers, but also take special care with the Isabelle/HOL maverick. Namely, we show that our local typedef proposal can be adapted to cope with Isabelle/HOL's type classes.

3 Proposal of a Logic Extension: Local Typedef

To address the limitation described in Section 1, we propose extending the HOL logic with a new rule for type definition with the following properties:

- It enables type definitions to be emulated inside proofs while avoiding the introduction of dependent types by a simple syntactic check.[2]
- It is natural and sound w.r.t. the standard HOL semantics à la Pitts [36], as well as consistent with the logic of Isabelle/HOL.

To motivate the formulation of the new rule and to understand the intuition behind it, we will first look deeper into the idea behind type definitions in HOL. Let us take a purely semantic perspective and ignore the rank-1 polymorphism for a minute. Then the principle behind type definitions simply states that for all types α and nonempty subsets A of them, there exists a type β isomorphic to A:

$$\forall \alpha.\, \forall a_{\alpha\ \mathsf{set}}.\, a \neq \emptyset \longrightarrow \exists \beta.\, \exists Abs_{\alpha\to\beta}\, Rep_{\beta\to\alpha}.\ {}_{\alpha}(\beta \approx a)^{Abs}_{Rep} \qquad (\star)$$

The typedef mechanism can be regarded as the result of applying a sequence of standard rules for connectives and quantifiers to $(\star)$ in a more expressive logic (notationally, we use Gentzen's sequent calculus):

1. Left $\forall$ rule, instantiating α and $a_{\sigma\ \mathsf{set}}$ with type σ and closed term A (of type $\sigma\ \mathsf{set}$) (both provided by the user), and left implication rule:

$$\dfrac{\dfrac{\Gamma \vdash A \neq \emptyset \qquad \Gamma, \exists \beta\, Abs\, Rep.\ {}_{\sigma}(\beta \approx A)^{Abs}_{Rep} \vdash \varphi}{\Gamma, (\star) \vdash \varphi}\ \forall_L, \forall_L, \longrightarrow_L}{\Gamma \vdash \varphi}\ \mathsf{Cut\ of}\ (\star)$$

2. Left $\exists$ rule for β, Abs and Rep, introducing some new/fresh type τ, and functions Abs^{τ} and Rep^{τ}:

$$\dfrac{\dfrac{\Gamma \vdash A \neq \emptyset \qquad \dfrac{\Gamma, {}_{\sigma}(\tau \approx A)^{\mathsf{Abs}^{\tau}}_{\mathsf{Rep}^{\tau}} \vdash \varphi}{\Gamma, \exists \beta\, Abs\, Rep.\ {}_{\sigma}(\beta \approx A)^{Abs}_{Rep} \vdash \varphi}\ \exists_L, \exists_L, \exists_L}{\Gamma, (\star) \vdash \varphi}\ \forall_L, \forall_L, \longrightarrow_L}{\Gamma \vdash \varphi}\ \mathsf{Cut\ of}\ (\star)$$

[2] Dependent type theory has its own pluses and minuses. Even if we came to the conclusion that the pluses prevail, we do not know how to combine dependent types with higher-order logic and the tools built around it. Hence the avoidance of the dependent types. Note that HOL-Omega does not include dependent types either.

The user further discharges $\Gamma \vdash A \neq \emptyset$, and therefore the overall effect of this chain is the sound addition of ${}_\sigma(\tau \approx A)^{\mathsf{Abs}^\tau}_{\mathsf{Rep}^\tau}$ as an extra assumption when trying to prove an arbitrary fact φ.

What we propose is to use a variant of the above with fewer instantiations, which takes better advantage of the result of Step 1: It refrains from introducing the type τ from Step 2, but keeps β as a fresh type variable. We obtain:

$$\dfrac{\Gamma \vdash A \neq \emptyset \qquad \dfrac{\dfrac{\Gamma, \exists \mathit{Abs}\ \mathit{Rep}.\ {}_\sigma(\beta \approx A)^{\mathit{Abs}}_{\mathit{Rep}} \vdash \varphi}{\Gamma, \exists \beta\, \mathit{Abs}\ \mathit{Rep}.\ {}_\sigma(\beta \approx A)^{\mathit{Abs}}_{\mathit{Rep}} \vdash \varphi}\ [\beta \text{ fresh}]\ \exists_L}{\Gamma, (\star) \vdash \varphi}\ \forall_L, \forall_L, \longrightarrow_L}{\Gamma \vdash \varphi}\ \text{Cut of } (\star)$$

To conclude, the overall rule, written (LT) as in "Local Typedef", looks as follows:

$$\dfrac{\Gamma \vdash A \neq \emptyset \qquad \Gamma \vdash (\exists \mathit{Abs}\ \mathit{Rep}.\ {}_\sigma(\beta \approx A)^{\mathit{Abs}}_{\mathit{Rep}}) \longrightarrow \varphi}{\Gamma \vdash \varphi}\ [\beta \text{ fresh for } A, \varphi, \Gamma]\ (\mathrm{LT})$$

This rule allows us to locally assume that there is a type β isomorphic to an arbitrary nonempty set A. The requirement that β be fresh for A prevents the introduction of a dependent type (since A may contain term variables).

The above discussion shows that (LT) is morally correct and, more importantly, *natural*, in the sense that it is an instance of a more general principle, namely the rule ($\star$). In the next section, we give a rigorous proof of the rule's consistency.

4 Consistency of Local Typedef

A typical development in HOL consists of issuing definitions and proving theorems from them. Thus, the only "axioms" that are introduced are the definitions. The appeal of this definitional approach to the many users of HOL is of course the guaranteed consistency. We will show that our new rule does not break the consistency of definitional theories. To this end, we first need to revisit the standard, model-theoretic argument for consistency.

4.1 Model Theory of HOL

We follow the model theory of HOL developed by Andrew Pitts [36]. Our presentation might diverge in concrete details and notation but the overall approach is the same.

We fix a Grothendieck universe $\mathcal{V}$, i.e., a transitive set that contains an infinite set and is closed under all standard set operations such as power set and union. It follows from transitivity and power-set closure that $\mathcal{V}$ is also closed under inclusion: if $A \in \mathcal{V}$ and $B \subseteq A$ then $B \in \mathcal{V}$. From this and the fact that $\mathcal{V}$ contains an infinite set, we obtain that $\mathcal{V}$ contains (copies of) $\mathbb{N}$ and $\mathbb{B} = \{\mathsf{false}, \mathsf{true}\}$.

We define our universe $\mathcal{U}$ as $\mathcal{U} = \mathcal{V} \setminus \emptyset$ (since we will interpret types as nonempty sets). Moreover, we fix a choice function, choice, that assigns to each set $A \in \mathcal{U}$ an element $\mathsf{choice}(A) \in A$.

We fix an interpretation function I of type constructors from the signature Σ, such that, for an n-ary type constructor $\kappa \in K$, $I(\kappa)$ is a function in $\mathcal{U}^n \to \mathcal{U}$. We assume the standard (fixed) interpretation for the built-in type constructors: $I(\mathsf{bool}) = \mathbb{B}$, $I(\mathsf{ind}) = \mathbb{N}$ and for the

function type constructor $\to$, we assume that $I(\to)$ sends any pair of sets (A, B) to $A \to B$ (the set of functions from A to B).

The interpretation of a type τ is a function $[\tau]_I : (\mathsf{TVar} \to \mathcal{U}) \to \mathcal{U}$, which takes an assignment to type variables $\theta : \mathsf{TVar} \to \mathcal{U}$ as input. Instead of $[\tau]_I(\theta)$, we will write $[\tau]_{I,\theta}$. The function is defined as follows:

$$[\alpha]_{I,\theta} = \theta(\alpha)$$
$$[(\sigma_1, \ldots, \sigma_n)\, \kappa]_{I,\theta} = I(\kappa)([\sigma_1]_{I,\theta}, \ldots, [\sigma_n]_{I,\theta}) \text{ where } \mathsf{arOf}(\kappa) = n$$

We fix an interpretation function for constants, also denoted by I, which assigns to any constant $c \in C$ a function $I(c) \in \prod_{\theta \in \mathsf{TVar} \to \mathcal{U}} [\mathsf{tpOf}(c)]_{I,\theta}$. We assume the standard (fixed) interpretation for the built-in constants: $I(\longrightarrow)(\theta)$ is the logical implication on $\mathbb{B}$; $I(=)(\theta)$ is the equality predicate in $\theta(\alpha) \to \theta(\alpha) \to \mathbb{B}$; $I(\mathsf{zero})(\theta)$ and $I(\mathsf{suc})(\theta)$ are the zero and successor on $\mathbb{N}$; the Hilbert choice operator is interpreted as

$$I(\varepsilon)(\theta)(f) = \begin{cases} \mathsf{choice}(\{a \in \theta(\alpha) \mid f(a) = \mathsf{true}\}) & \text{if the set is non-empty,} \\ \mathsf{choice}(\theta(\alpha)) & \text{otherwise.} \end{cases}$$

We say that an assignment $\xi : \mathsf{Var} \to \mathcal{U}$ (of term variables to elements of $\mathcal{U}$) is θ-compatible if $\xi \in \prod_{x_\tau \in \mathsf{Var}} [\tau]_{I,\theta}$, i.e., $\xi(x_\tau) \in [\tau]_{I,\theta}$ for all term variables x_τ. The interpretation of a term $t : \sigma$ is a function

$$[t]_I \in \prod_{\theta \in \mathsf{TVar} \to \mathcal{U}} \left(\prod_{x_\tau \in \mathsf{Var}} [\tau]_{I,\theta} \right) \to [\sigma]_{I,\theta}.$$

Thus, the interpretation of terms takes an assignment to type variables θ and a (θ-compatible) assignment to term variables ξ as input. We will write $[t]_{I,\theta,\xi}$ instead of $[t]_I(\theta)(\xi)$. We define $[t]_{I,\theta,\xi}$ as follows:

$$[x_\sigma]_{I,\theta,\xi} = \xi(x_\sigma)$$

$$[c_\sigma]_{I,\theta,\xi} = I(c)(\theta') \text{ where we obtain}^3 \ \rho \text{ s.t. } \sigma \leq_\rho \mathsf{tpOf}(c) \text{ and define } \theta'(\alpha) = [\rho(\alpha)]_{I,\theta}$$

$$[t_1\ t_2]_{I,\theta,\xi} = [t_1]_{I,\theta,\xi}\ [t_2]_{I,\theta,\xi}$$

$$[\lambda x_\sigma.\ t]_{I,\theta,\xi} = \bigwedge_{a \in [\sigma]_{I,\theta}} [t]_{I,\theta,\xi[x_\sigma \leftarrow a]}$$

Above, the interpretation of lambda abstraction is the function sending each $a \in [\sigma]_{I,\theta}$ to $[t]_{I,\theta,\xi[x_\sigma \leftarrow a]}$, where $\xi[x_\sigma \leftarrow a]$ denotes ξ updated with a at x_σ.

Thus, an interpretation I consists of two homonymous functions, one for type constructors and one for constants. We say that I satisfies a sequent (Γ, φ), written $\Gamma \vDash_I \varphi$, if for all θ and ξ it holds that $[\varphi]_{I,\theta,\xi} = \mathsf{true}$ whenever $[\psi]_{I,\theta,\xi} = \mathsf{true}$ for all $\psi \in \Gamma$. If Γ is empty, we write $\vDash_I \varphi$. A deduction rule has the form $\frac{(\Gamma_1,\varphi_1) \ldots (\Gamma_n,\varphi_n)}{(\Gamma,\varphi)}$; we say that such a deduction rule is sound for I if $\Gamma \vDash_I \varphi$ holds whenever $\Gamma_j \vDash_I \varphi_j$ holds for all $j \in \{1, \ldots, n\}$.

We say that a interpretation I is a model of a theory D if I satisfies all HOL axioms and all formulas from D. Pitts proved that (i) the deduction rules of HOL are sound for all standard interpretations, and (ii) every definitional theory has a standard model. The facts (i) and (ii) give us that, for every definitional theory D, if $D; \Gamma \vdash \varphi$, then there exists a standard model I of D such that $\Gamma \vDash_I \varphi$. In particular, this implies the consistency of the HOL definitional theories, since $\nvDash_I \mathsf{False}$.

[3] There is always such ρ since we work with well-typed terms and moreover it is unique.

4.2 Proof of Consistency

Now we have enough background material to carry out the proof of consistency of (LT) in HOL. We shall employ the following folklore lemma, stating that interpretations only depend on the free variables:

Lemma 1 Let I be an interpretation.
(1) If θ and θ' do not differ on $\mathsf{TV}(\tau)$, then $[\tau]_{I,\theta} = [\tau]_{I,\theta'}$.
(2) If θ and θ' do not differ on $\mathsf{TV}(t)$ and ξ and ξ' do not differ on $\mathsf{FV}(t)$, then $[t]_{I,\theta,\xi} = [t]_{I,\theta',\xi'}$.

Lemma 2 The rule (LT) is sound for any standard interpretation.

Proof. Let us fix a standard interpretation I and assume that the hypotheses of the (LT) rule are satisfied by I, i.e.,

$$\Gamma \vDash_I A \neq \emptyset \tag{3}$$

$$\Gamma \vDash_I (\exists \mathit{Abs}\ \mathit{Rep}.\ {}_\sigma(\beta \approx A)^{\mathit{Abs}}_{\mathit{Rep}}) \longrightarrow \varphi \tag{4}$$

We wish to prove the rule's conclusion, $\Gamma \vDash_I \varphi$. To this end, let us fix θ and a θ-compatible ξ such that $[\psi]_{I,\theta,\xi} = \mathsf{true}$ for all $\psi \in \Gamma$. We wish to prove $[\varphi]_{I,\theta,\xi} = \mathsf{true}$.

Let us temporarily fix $B \in \mathcal{U}$, and let θ_B denote $\theta[\beta \leftarrow B]$ (θ with β updated to B). We let ξ_B be a θ_B-compatible assignment to term variables such that ξ_B is the same as ξ on the free variables of A, Γ and φ. Since β is fresh for Γ, by Lemma 1 we have $[\psi]_{I,\theta_B,\xi_B} = \mathsf{true}$ for all $\psi \in \Gamma$. Using (3) and (4) and under the standard interpretation of $\longrightarrow$ in (4), we obtain:

$$[A \neq \emptyset]_{I,\theta_B,\xi_B} = \mathsf{true}$$

$$[\exists \mathit{Abs}\ \mathit{Rep}.\ {}_\sigma(\beta \approx A)^{\mathit{Abs}}_{\mathit{Rep}}]_{I,\theta_B,\xi_B} = \mathsf{true}\ \text{ implies }\ [\varphi]_{I,\theta_B,\xi_B} = \mathsf{true}.$$

Since β is fresh for A and φ, with Lemma 1 we obtain:

$$[A \neq \emptyset]_{I,\theta,\xi} = \mathsf{true}, \tag{5}$$

$$[\exists \mathit{Abs}\ \mathit{Rep}.\ {}_\sigma(\beta \approx A)^{\mathit{Abs}}_{\mathit{Rep}}]_{I,\theta_B,\xi_B} = \mathsf{true}\ \text{ implies }\ [\varphi]_{I,\theta,\xi} = \mathsf{true}. \tag{6}$$

Since B above was arbitrary, if we were able to find *some* $B \in \mathcal{U}$ such that the antecedent of (6), namely,

$$[\exists \mathit{Abs}\ \mathit{Rep}.\ {}_\sigma(\beta \approx A)^{\mathit{Abs}}_{\mathit{Rep}}]_{I,\theta_B,\xi_B} = \mathsf{true} \tag{7}$$

holds, then the proof would be finished, since we would obtain $[\varphi]_{I,\theta,\xi} = \mathsf{true}$, as desired.

From (5) we obtain

$$\{a \in [\sigma]_{I,\theta} \mid [A]_{I,\theta,\xi}\ a = \mathsf{true}\} \neq \emptyset. \tag{8}$$

Recall that $\mathcal{V}$ is closed under inclusions and thus $\mathcal{U}$ is closed under inclusions of nonempty subsets. To prove (7), we take B to be $\{a \in [\sigma]_{I,\theta} \mid [A]_{I,\theta,\xi}\ a = \mathsf{true}\}$ and since $B \subseteq [\sigma]_{I,\theta}$, $[\sigma]_{I,\theta} \in \mathcal{U}$ and (8), we obtain $B \in \mathcal{U}$. Note that, again by Lemma 1, we have $[A]_{I,\theta_B,\xi_B} = [A]_{I,\theta,\xi}$ and, since $\mathsf{TV}(\sigma) \subseteq \mathsf{TV}(A)$, we have $[\sigma]_{I,\theta_B} = [\sigma]_{I,\theta}$. We define $\mathsf{Abs} : [\sigma]_{I,\theta} \to B$ as

$$\mathsf{Abs}(x) = \begin{cases} x & \text{if } x \in B \\ \epsilon(B) & \text{otherwise} \end{cases}$$

and $\mathsf{Rep} : B \to [\sigma]_{I,\theta}$ as the inclusion map. It is a routine to verify

$$[{}_\sigma(\beta \approx A)^{\mathit{Abs}}_{\mathit{Rep}}]_{I,\theta_B,\xi_B[\mathit{Abs} \leftarrow \mathsf{Abs}, \mathit{Rep} \leftarrow \mathsf{Rep}]} = \mathsf{true},$$

which proves (7). □

The bottom line of the above proof was to show a semantic analog of ($\star$), our intuitive formulation of HOL's typedef in a logic richer than HOL: Assuming (8) holds, we find B such that (7) holds. The important twist is that this was performed in an arbitrary proof context, freed from the restrictions imposed by traditional HOL.

Proposition 1 Any HOL definitional theory is consistent w.r.t. HOL deduction extended with the (LT) rule.

Proof. Let D be a definitional theory. Assume, for a contradiction, that $D; \vdash' \mathsf{False}$ where $\vdash'$ denotes HOL deduction enriched with (LT). We take I to be a model of the HOL axioms (of whose existence we know from Pitts). Since the HOL rules are sound for I and, from the above lemma, (LT) is also sound for I, we obtain $\models_I \mathsf{False}$, which is impossible. □

Consistency also holds for the case of Isabelle/HOL:

Proposition 2 Any Isabelle/HOL definitional theory is consistent w.r.t. HOL deduction extended with the (LT) rule.

The proof of this proposition can be found in our recent paper on proving Isabelle/HOL's consistency [28]. There, we first propose a stronger logic of HOL with comprehension types (HOLC) and show that HOLC is consistent. Then we soundly translate Isabelle/HOL's logic into HOLC and prove that the (LT) rule is admissible in HOLC, which makes it consistent in Isabelle/HOL [28, §4].

5 From Types to Sets in HOL

Let us look again at the motivating example from Section 1 and see how the rule (LT) allows us to achieve the relativization from a type-based theorem to a set-based theorem in HOL or Isabelle/HOL without type classes. We assume (1) is a theorem, and wish to prove (2). We fix α and $A_{\alpha\,\mathsf{set}}$ and assume $A \neq \emptyset$. Applying (LT), we obtain a type β (represented by a fresh type variable) such that $\exists \mathit{Abs}\ \mathit{Rep}.\ {}_{\alpha}(\beta \approx A)^{\mathit{Abs}}_{\mathit{Rep}}$, from which we obtain Abs and Rep such that ${}_{\alpha}(\beta \approx A)^{\mathsf{Abs}}_{\mathsf{Rep}}$. From this, (1) with α instantiated to β, and the definition of lists, we obtain

$$\exists xs_{\beta\,\mathsf{list}} \in \mathsf{lists}\,(\mathsf{UNIV}_{\beta\,\mathsf{set}}).\ \mathsf{P}_{\beta\,\mathsf{list}\to\mathsf{bool}}\ xs.$$

Furthermore, using that Abs and Rep are isomorphisms between $A_{\alpha\,\mathsf{set}}$ and $\mathsf{UNIV}_{\beta\,\mathsf{set}}$, we obtain

$$\exists xs_{\alpha\,\mathsf{list}} \in \mathsf{lists}\ A_{\alpha\,\mathsf{set}}.\ \mathsf{P}_{\alpha\,\mathsf{list}\to\mathsf{bool}}\ xs,$$

as desired.[4]

We will consider a general case now. Let us start with a type-based theorem

$$\forall \alpha.\ \varphi[\alpha], \tag{9}$$

[4] In order for this to work, we have silently assumed a connection between $\mathsf{P}_{\beta\,\mathsf{list}\to\mathsf{bool}}$ and $\mathsf{P}_{\alpha\,\mathsf{list}\to\mathsf{bool}}$, namely that P is parametric w.r.t. injection. More precisely that for every binary relation $R_{\alpha\to\beta\to\mathsf{bool}}$ defining an injection of β into α and for every two lists $xs_{\alpha\,\mathsf{list}}$ and $ys_{\beta\,\mathsf{list}}$ whose elements are pairwise related by R, it holds that $\mathsf{P}\ xs$ if and only if $\mathsf{P}\ ys$—see Section 7 for more on parametricity.

where $\varphi[\alpha]$ is a formula containing α. We fix α and $A_{\alpha\,\mathsf{set}}$, assume $A \neq \emptyset$ and "define" a new type β isomorphic to A. Technically, we fix a fresh type variable β and assume

$$\exists Abs\ Rep.\ {}_{\alpha}(\beta \approx A)^{Abs}_{Rep}. \tag{10}$$

From the last formula, we can obtain the isomorphism Abs and Rep between β and A. Having the isomorphisms, we can carry out the relativization along them and prove

$$\varphi[\beta] \longleftrightarrow \varphi^{\mathsf{on}}[\alpha, A_{\alpha\,\mathsf{set}}], \tag{11}$$

where $\varphi^{\mathsf{on}}[\alpha, A_{\alpha\,\mathsf{set}}]$ is the relativization of $\varphi[\beta]$. In the motivational example:

$$\varphi[\beta] = \exists xs_{\beta\,\mathsf{list}}.\ \mathsf{P}\ xs$$
$$\varphi^{\mathsf{on}}[\alpha, A_{\alpha\,\mathsf{set}}] = \exists xs_{\alpha\,\mathsf{list}} \in \mathsf{lists}\ A.\ \mathsf{P}\ xs$$

We postpone the discussion of how to derive φ^{on} from φ in a principled way and how to automatically prove the equivalence between them until Section 7. Here, we only appeal to the intuition: For example, if φ contains the universal quantification $\forall x_\beta$, we replace it by the related bounded quantification $\forall x_\alpha \in A$ in φ^{on}. Or, if φ contains the predicate $\mathsf{inj}\ f_{\beta\to\gamma}$, we replace it by the related notion of $\mathsf{inj}^{\mathsf{on}}\ A_{\alpha\,\mathsf{set}}\ f_{\alpha\to\gamma}$ in φ^{on}.

Since the left-hand side of the equivalence (11) is an instance of (9), we discharge the left-hand side and obtain $\varphi^{\mathsf{on}}[\alpha, A_{\alpha\,\mathsf{set}}]$, which no longer contains the locally "defined" type β. Thus we can discard β. Technically, we use the (LT) rule and remove the assumption (10). Thus we obtain the final result:

$$\forall\alpha.\ \forall A_{\alpha\,\mathsf{set}}.\ A \neq \emptyset \longrightarrow \varphi^{\mathsf{on}}[\alpha, A]$$

This theorem is the set-based version of $\forall\alpha.\ \varphi[\alpha]$.

We will move to Isabelle/HOL in the next section and explore how the isomorphic journey between types and sets proceeds in the environment where we are allowed to restrict type variables by type-class annotations.

6 From Types to Sets in Isabelle/HOL

Isabelle/HOL goes beyond traditional HOL, extending it by ad hoc overloading and axiomatic type classes. In this section, we explain how these features are in conflict with the algorithm described in Section 5 and show how to circumvent the problem.

6.1 Local Axiomatic Type Classes

The first complication is the implicit assumptions on types given by the axiomatic type classes. Recall that α_{finite} means that α can be instantiated only with a type that we proved to fulfill the conditions of the type class finite, namely, that the type must contain finitely many elements.

To explain the complication on an example, let us modify (9) to speak about types of class finite:

$$\forall\alpha_{\mathsf{finite}}.\ \varphi[\alpha_{\mathsf{finite}}] \tag{12}$$

As a modification of the algorithm from Section 5, we fix a set A and assume that it is nonempty *and finite*. As before, we locally define a new type β isomorphic to A. Although β fulfills the condition of the type class finite, we cannot add the type into the type class since this action is allowed only at the global theory level in Isabelle and not locally in a proof context.

On the other hand, without adding β into finite we cannot continue since we need to instantiate β for α_{finite} to prove the analog of the equivalence (11). Our solution is to internalize the type-class assumption in (12) and obtain

$$\forall\alpha.\ \mathsf{finite}(\alpha) \longrightarrow \varphi[\alpha], \tag{13}$$

where $\mathsf{finite}(\alpha)$ is a term of type bool, which is true if and only if α is a finite type.[5] Now we can instantiate α by β and get $\mathsf{finite}(\beta) \longrightarrow \varphi[\beta]$. Using the fact that the relativization of $\mathsf{finite}(\beta)$ is $\mathsf{finite}\ A$, we apply the isomorphic translation between β and A and obtain

$$\mathsf{finite}\ A \longrightarrow \varphi^{\mathsf{on}}[\alpha, A].$$

Quantifying over the fixed variables and adding the assumptions yields the final result, the set-based version of (12):

$$\forall\alpha.\ \forall A_{\alpha\ \mathsf{set}}.\ A \neq \emptyset \longrightarrow \mathsf{finite}\ A \longrightarrow \varphi^{\mathsf{on}}[\alpha, A]$$

The internalization of type classes (inferring (13) from (12)) is already supported by the kernel of Isabelle—thus no further work is required from us. The rule for internalization of type classes is a result of the work by Haftmann and Wenzel [16, 42].

6.2 Local Overloading

In the previous section we addressed implicit assumptions on types given by axiomatic type classes and showed how to reduce the relativization of such types to the original translation algorithm by internalizing the type classes as predicates on types. As we explained in Section 2.3, the mechanism of Haskell-like type classes in Isabelle is more general than the notion of axiomatic type classes since additionally we are allowed to associate operations with every type class. In this respect, the type class finite is somewhat special since there are no operations associated with it.

Therefore we use semigroups as the running example in this section since semigroups require an associated operation—multiplication. A general specification of a semigroup would contain a nonempty set $A_{\alpha\ \mathsf{set}}$, a binary operation $f_{\alpha\to\alpha\to\alpha}$ such that A is closed under f, and a proof of the specific property of semigroups that f is associative on A. We capture the last property by the predicate

$$\mathsf{semigroup}^{\mathsf{on}}_{\mathsf{with}}\ A\ f = (\forall x\ y\ z \in A.\ f\ (f\ x\ y)\ z = f\ x\ (f\ y\ z)),$$

which we read along the paradigm: *a structure* on *the set* A with *operations* $f_1, \ldots, f_n$.

[5] This is Wenzel's approach [42] to represent axiomatic type classes by internalizing them as predicates on types, i.e., constants of type $\forall\alpha.$ bool. As this particular type is not allowed in Isabelle, Wenzel uses instead α itself $\to$ bool, where α itself is a singleton type.

The realization of semigroups by type classes in Isabelle is an example that reflects the problem's full generality. The type σ can belong to the type class $\mathsf{semigroup}$ if $\mathsf{semigroup}(\sigma)$ is provable, where

$$\mathsf{semigroup}(\alpha) \text{ iff } \forall x_\alpha\ y_\alpha\ z_\alpha.\ (x * y) * z = x * (y * z). \tag{14}$$

Notice that the associated multiplication operation is represented by the *global* overloaded constant $*_{\alpha\to\alpha\to\alpha}$, which will cause the complication.

Let us now relativize $\forall\alpha_{\mathsf{semigroup}}.\ \varphi[\alpha_{\mathsf{semigroup}}]$. We fix a nonempty set A, a binary f such that A is closed under f and assume $\mathsf{semigroup}^{\mathsf{on}}_{\mathsf{with}}\ A\ f$. As before, we locally define β to be isomorphic to A and obtain the respective isomorphisms Abs and Rep.

Having defined β, we want to prove that β belongs into $\mathsf{semigroup}$. Using the approach from the previous section, this goal translates into proving $\mathsf{semigroup}(\beta)$, which requires that the overloaded constant $*_{\beta\to\beta\to\beta}$ used in the definition of $\mathsf{semigroup}$ (see (14)) must be isomorphic to f on A. In other words, we have to locally define $*_{\beta\to\beta\to\beta}$ to be a projection of f onto β, i.e., $x_\beta * y_\beta$ must equal $\mathsf{Abs}(f\ (\mathsf{Rep}\ x)\ (\mathsf{Rep}\ y))$. Although we can locally "define" a new constant (fix a fresh term variable c and assume $c = t$), we cannot overload the global symbol $*$ locally for β. This is not supported by Isabelle.

We will cope with the complication by compiling out the overloaded constant $*$ from

$$\forall\alpha.\ \mathsf{semigroup}(\alpha) \longrightarrow \varphi[\alpha] \tag{15}$$

as follows: Whenever $c = \ldots * \ldots$ (i.e., c was defined in terms of $*$ and thus depends implicitly on the overloaded meaning of $*$), define $c_{\mathsf{with}}\ f = \ldots f \ldots$ and use it instead of c. The parameter f abstracts over the meaning of $*$ here: whenever we want to use c_{with}, we have to explicitly specify how to perform multiplication in c_{with} by instantiating f. That is to say, the implicit meaning of $*$ in c was made explicit by f in c_{with}. Using this approach, we obtain:

$$\forall\alpha.\ \forall f_{\alpha\to\alpha\to\alpha}.\ \mathsf{semigroup}_{\mathsf{with}}\ f \longrightarrow \varphi_{\mathsf{with}}[\alpha, f], \tag{16}$$

where $\mathsf{semigroup}_{\mathsf{with}}\ f_{\alpha\to\alpha\to\alpha} = (\forall x_\alpha\ y_\alpha\ z_\alpha.\ f\ (f\ x\ y)\ z = f\ x\ (f\ y\ z))$ and similarly for φ_{with}. For now, let us assume that we can provably obtain (16) from (15) and let us look at how it helps us to finish the relativization and later we will explain how to derive (16) as a theorem.

Given (16), we will instantiate α with β and obtain

$$\forall f_{\beta\to\beta\to\beta}.\ \mathsf{semigroup}_{\mathsf{with}}\ f \longrightarrow \varphi_{\mathsf{with}}[\beta, f].$$

Recall that the quantification over all functions of type $\beta\to\beta\to\beta$ is isomorphic to the bounded quantification over all functions of type $\alpha\to\alpha\to\alpha$ under which $A_{\alpha\,\mathsf{set}}$ is closed.[6] The difference after compiling out the overloaded constant $*$ is that now we are isomorphically relating two bounded (local) variables from the quantification and not a global constant $*$ to a local variable.

Thus we reduced the relativization once again to the original algorithm and can obtain the set-based version

$$\begin{aligned}&\forall\alpha.\ \forall A_{\alpha\,\mathsf{set}}.\ A \neq \emptyset \longrightarrow\\ &\quad \forall f_{\alpha\to\alpha\to\alpha}.\ (\forall x_\alpha\ y_\alpha \in A.\ f\ x\ y \in A) \longrightarrow \mathsf{semigroup}^{\mathsf{on}}_{\mathsf{with}}\ A\ f \longrightarrow \varphi^{\mathsf{on}}_{\mathsf{with}}[\alpha, A, f].\end{aligned}$$

[6] Let us recall that $\forall x.\ P\ x$ is a shorthand for $\mathsf{All}\ (\lambda x.\ P\ x)$ and $\forall x \in A.\ P\ x$ for $\mathsf{Ball}\ A\ (\lambda x.\ P\ x)$, where All and Ball are the HOL constants for quantification. Thus the statement about isomorphism between the two quantifications means isomorphism between All and $\mathsf{Ball}\ A$.

Let us discuss how to obtain (16) as a theorem. We derive it by performing the *dictionary construction*, whose detailed description can be found, for example, in the paper by Krauss and Schropp [26, §5.2]. We will outline the process only informally here. Our task is to compile out an overloaded constant $*$ from a term s. As a first step, we transform s into $s_{\mathsf{with}}[*/f]$ such that $s = s_{\mathsf{with}}[*/f]$ and such that unfolding the definitions of all constants in s_{with} does not yield $*$ as a subterm. We proceed for every constant c in s as follows: if c has no definition, we do not do anything. If c was defined as $c = t$, we first apply the construction recursively on t and obtain t_{with} such that $t = t_{\mathsf{with}}[*/f]$; thus $c = t_{\mathsf{with}}[*/f]$. Now we define a new constant $c_{\mathsf{with}}\ f = t_{\mathsf{with}}$. As $c_{\mathsf{with}}\ * = c$, we replace c in s by $c_{\mathsf{with}}\ *$. At the end, we obtain $s = s_{\mathsf{with}}[*/f]$ as a theorem. Notice that this procedure produces s_{with} that does not semantically depend on $*$ only if there is no type in s that depends on $*$.

Thus the above-described step applied to (15) produces

$$\forall\alpha.\ \mathsf{semigroup}_{\mathsf{with}}\ *_{\alpha\to\alpha\to\alpha} \longrightarrow \varphi_{\mathsf{with}}[\alpha, f_{\alpha\to\alpha\to\alpha}][*_{\alpha\to\alpha\to\alpha}/f_{\alpha\to\alpha\to\alpha}].$$

To finish the dictionary construction, we replace every occurrence of $*_{\alpha\to\alpha\to\alpha}$ by a universally quantified variable $f_{\alpha\to\alpha\to\alpha}$ and obtain (16). This derivation step is not currently allowed in Isabelle. The idea why this is a sound derivation is as follows: since $*_{\alpha\to\alpha\to\alpha}$ is a type-class operation, there exist overloaded definitions only for strict instances of $*$ (such as $*_{\mathsf{nat}\to\mathsf{nat}\to\mathsf{nat}}$) but never for $*_{\alpha\to\alpha\to\alpha}$; thus the meaning of $*_{\alpha\to\alpha\to\alpha}$ remains unrestricted. That is to say, $*_{\alpha\to\alpha\to\alpha}$ permits any interpretation and hence it must behave as a term variable. We will formulate a rule (an extension of Isabelle's logic) that allows us to perform the above-described derivation.

First, let us recall that $\rightsquigarrow^{\downarrow}$ is the substitutive closure of the constant/type dependency relation $\rightsquigarrow$ from Section 2.3 and Δ_c is the set of all types for which c was overloaded. The notation $\sigma \not\leq S$ means that σ is not an instance of any type in S. We shall write R^+ for the transitive closure of R. Now we can formulate the Unoverloading Rule (UO):

$$\frac{\varphi[c_\sigma/x_\sigma]}{\forall x_\sigma.\ \varphi}\ [\neg(u \rightsquigarrow^{\downarrow+} c_\sigma) \text{ for any type or constant } u \text{ in } \varphi;\ \sigma \not\leq \Delta_c] \quad \text{(UO)}$$

This means that we can replace occurrences of the constant c_σ in φ by the universally quantified variable x_σ under the following two side conditions:

1. All types and constant instances in φ do not semantically depend on c_σ through a chain of constant and type definitions. The constraint is fulfilled in the first step of the dictionary construction since for example $\varphi_{\mathsf{with}}[\alpha, *]$ does not contain any hidden $*$s due to the construction of φ_{with}.[7]
2. There is no matching definition for c_σ. In our use case, c_σ is always a type-class operation with its most general type (e.g., $*_{\alpha\to\alpha\to\alpha}$). As already mentioned, we overload a type-class operation only for strictly more specific types (such as $*_{\mathsf{nat}\to\mathsf{nat}\to\mathsf{nat}}$) and never for its most general type and thus the condition $\sigma \not\leq \Delta_c$ must be fulfilled.

Proposition 3 Any Isabelle/HOL definitional theory is consistent w.r.t. HOL deduction extended with the (LT) and (UO) rules.

Again, the proof of this proposition is based on the translation from Isabelle/HOL to HOL with comprehension types (HOLC) from [28]. It is presented in [28, §4].

Notice that the (UO) rule suggests that even in presence of *ad hoc* overloading, the polymorphic overloaded constants retain parametricity under some conditions. In the next section, we will look at a concrete example of relativization of a formula with type classes.

[7] Unless there is a type depending on $*$.

6.3 Example: Relativization of Topological Spaces

We will show an example of relativization of a type-based theorem with type classes in a set-based theorem from the field of topology (addressing Immler's concern discussed in Section 1). The type class in question will be a topological space, which has one associated operation $\mathsf{open} : \alpha\ \mathsf{set} \to \mathsf{bool}$, a predicate defining the open subsets of α. We require that the whole space is open, finite intersections of open sets are open, finite or infinite unions of open sets are open and that every two distinct points can be separated by two open sets that contain them. Such a topological space is called a T2 space and therefore we call the respective type class $\mathsf{T2\text{-}space}$.

One of the basic properties of T2 spaces is the fact that every compact set is closed:

$$\forall \alpha_{\mathsf{T2\text{-}space}}.\ \forall S_{\alpha\ \mathsf{set}}.\ \mathsf{compact}\ S \longrightarrow \mathsf{closed}\ S \tag{17}$$

A set is compact if every open cover of it has a finite subcover. A set is closed if its complement is open. i.e., $\mathsf{closed}\ S = \mathsf{open}\ (-S)$. Recall that our main motivation is to solve the problem when we have a T2 space on a proper subset of α. Let us show the translation of (17) into a set-based variant, which solves the problem. We will observe what happens to the predicate closed during the translation.

We will first internalize the type class $\mathsf{T2\text{-}space}$ and then abstract over its operation open via the first step of the dictionary construction. As a result, we obtain

$$\forall \alpha.\ \mathsf{T2\text{-}space}_{\mathsf{with}}\ \mathsf{open} \longrightarrow \forall S_{\alpha\ \mathsf{set}}.\ \mathsf{compact}_{\mathsf{with}}\ \mathsf{open}\ S \longrightarrow \mathsf{closed}_{\mathsf{with}}\ \mathsf{open}\ S,$$

where $\mathsf{closed}_{\mathsf{with}}\ \mathit{open}\ S = \mathit{open}\ (-S)$. Let us apply (UO) and generalize over open:

$$\begin{aligned}&\forall \alpha.\ \forall \mathit{open}_{\alpha\ \mathsf{set}\to\mathsf{bool}}.\\ &\quad \mathsf{T2\text{-}space}_{\mathsf{with}}\ \mathit{open} \longrightarrow \forall S_{\alpha\ \mathsf{set}}.\ \mathsf{compact}_{\mathsf{with}}\ \mathit{open}\ S \longrightarrow \mathsf{closed}_{\mathsf{with}}\ \mathit{open}\ S\end{aligned} \tag{18}$$

The last formula is a variant of (17) after we internalized the type class $\mathsf{T2\text{-}space}$ and compiled out its operation. Now we reduced the task to the original algorithm (using Local Typedef) from Section 5. As always, we fix a nonempty set $A_{\alpha\ \mathsf{set}}$, locally define β to be isomorphic to A and transfer the β-instance of (18) onto the $A_{\alpha\ \mathsf{set}}$-level:

$$\begin{aligned}&\forall \alpha.\ \forall A_{\alpha\ \mathsf{set}}.\ A \neq \emptyset \longrightarrow \forall \mathit{open}_{\alpha\ \mathsf{set}\to\mathsf{bool}}.\ \mathsf{T2\text{-}space}^{\mathsf{on}}_{\mathsf{with}}\ A\ \mathit{open} \longrightarrow\\ &\quad \forall S_{\alpha\ \mathsf{set}} \subseteq A.\ \mathsf{compact}^{\mathsf{on}}_{\mathsf{with}}\ A\ \mathit{open}\ S \longrightarrow \mathsf{closed}^{\mathsf{on}}_{\mathsf{with}}\ A\ \mathit{open}\ S\end{aligned}$$

This is the set-based variant of the original theorem (17). Let us show what happened to $\mathsf{closed}_{\mathsf{with}}$: its relativization is defined as $\mathsf{closed}^{\mathsf{on}}_{\mathsf{with}}\ A\ \mathit{open}\ S = \mathit{open}\ (-S \cap A)$. Notice that we did not have to restrict *open* while moving between β and A (since the function does not produce any values of type β), whereas S is restricted since subsets of β correspond to subsets of A.

6.4 General Case

Having seen a concrete example, let us finally aim for the general case. Let us assume that Υ is a type class depending on the overloaded constants $*_1, \ldots, *_n$, written $\overline{*}$. We write $A \downarrow \overline{f}$ to mean that A is closed under operations $f_1, \ldots, f_n$.

The following derivation tree shows how we derive, from the type-based theorem $\vdash \forall \alpha\Upsilon.\ \varphi[\alpha\Upsilon]$ (the topmost formula in the tree), its set-based version (the bottommost formula). Explanation of the derivation steps follows after the tree.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash \forall \alpha\gamma.\ \varphi[\alpha\gamma]}{\vdash \forall \alpha.\ \Upsilon(\alpha) \longrightarrow \varphi[\alpha]}\ (1)}{\vdash \forall \alpha.\ \Upsilon_{\mathsf{with}}\ \overline{*}[\alpha] \longrightarrow \varphi_{\mathsf{with}}[\alpha, \overline{f}][\overline{*}/\overline{f}]}\ (2)}{\vdash \forall \alpha.\ \forall \overline{f}[\alpha].\ \Upsilon_{\mathsf{with}}\ \overline{f} \longrightarrow \varphi_{\mathsf{with}}[\alpha, \overline{f}]}\ (3)}{A_\alpha\ \mathsf{set} \neq \emptyset,\ {}_\alpha(\beta \approx A)^{Abs}_{Rep} \vdash \forall \alpha.\ \forall \overline{f}[\alpha].\ \Upsilon_{\mathsf{with}}\ \overline{f} \longrightarrow \varphi_{\mathsf{with}}[\alpha, \overline{f}]}\ (4)}{A_\alpha\ \mathsf{set} \neq \emptyset,\ {}_\alpha(\beta \approx A)^{Abs}_{Rep} \vdash \forall \overline{f}[\beta].\ \Upsilon_{\mathsf{with}}\ \overline{f} \longrightarrow \varphi_{\mathsf{with}}[\beta, \overline{f}]}\ (5)}{A_\alpha\ \mathsf{set} \neq \emptyset,\ {}_\alpha(\beta \approx A)^{Abs}_{Rep} \vdash \forall \overline{f}[\alpha].\ A \downarrow \overline{f} \longrightarrow \Upsilon^{\mathsf{on}}_{\mathsf{with}}\ A\ \overline{f} \longrightarrow \varphi^{\mathsf{on}}_{\mathsf{with}}[\alpha, A, \overline{f}]}\ (6)}{A_\alpha\ \mathsf{set} \neq \emptyset \vdash \forall \overline{f}[\alpha].\ A \downarrow \overline{f} \longrightarrow \Upsilon^{\mathsf{on}}_{\mathsf{with}}\ A\ \overline{f} \longrightarrow \varphi^{\mathsf{on}}_{\mathsf{with}}[\alpha, A, \overline{f}]}\ (7)}{\vdash \forall \alpha.\ \forall A_\alpha\ \mathsf{set}.\ A \neq \emptyset \longrightarrow \forall \overline{f}[\alpha].\ A \downarrow \overline{f} \longrightarrow \Upsilon^{\mathsf{on}}_{\mathsf{with}}\ A\ \overline{f} \longrightarrow \varphi^{\mathsf{on}}_{\mathsf{with}}[\alpha, A, \overline{f}]}\ (8)$$

Derivation steps:

(1) The class internalization from Section 6.1.
(2) The first step of the dictionary construction from Section 6.2.
(3) The Unoverloading rule (UO) from Section 6.2.
(4) We fix fresh α, $A_\alpha\ \mathsf{set}$ and assume that A is nonempty. We locally define a new type β to be isomorphic to A; i.e., we fix fresh β, $Abs_{\alpha\to\beta}$ and $Rep_{\beta\to\alpha}$ and assume ${}_\alpha(\beta \approx A)^{Abs}_{Rep}$.
(5) We instantiate α in the conclusion with β.
(6) Relativization along the isomorphism between β and A—see Section 7.
(7) Since *Abs* and *Rep* are present only in ${}_\alpha(\beta \approx A)^{Abs}_{Rep}$, we can existentially quantify over them and replace the hypothesis with $\exists Abs\ Rep.\ {}_\alpha(\beta \approx A)^{Abs}_{Rep}$, which we discharge by the Local Typedef rule from Section 3, as β is not present elsewhere either (the previous step (6) removed all occurrences of β in the conclusion).
(8) We move all hypotheses into the conclusion and quantify over all fixed variables.

As previously discussed, step (2), the dictionary construction, cannot be performed for types depending on overloaded constants unless we want to compile out such types too. In the next section, we will explain the last missing piece: the relativization step (6).

Note that our approach addresses one of the long-standing user complaints: the impossibility to provide two different orders for the same type when using the type class of orders. With our approach, users can still enjoy the advantages of type classes while proving abstract properties about orders, and then only export the final product as a set-based theorem (which quantifies over all possible orders).

7 Transfer: Automated Relativization

In this section, we will describe a procedure that automatically achieves relativization of the type-based theorems. Recall that we are facing the following problem: we have two types β and α such that β is isomorphic to some (nonempty) set $A_\alpha\ \mathsf{set}$, a proper subset of α, via two isomorphisms $\mathsf{Abs}_{\alpha\to\beta}$ and $\mathsf{Rep}_{\beta\to\alpha}$. In this setting, given a formula $\varphi[\beta]$, we want to find its isomorphic image $\varphi^{\mathsf{on}}[\alpha, A]$ along Abs and Rep and prove $\varphi[\beta] \longleftrightarrow \varphi^{\mathsf{on}}[\alpha, A]$. Thanks to the previous work in which the first author of this paper participated [23], we can use Isabelle's Transfer tool, which automatically synthesizes the relativized formula $\varphi^{\mathsf{on}}[\alpha, A]$ and proves the equivalence with the original formula $\varphi[\beta]$.

We will sketch the main principles of the tool on the following example, where (20) is a relativization of (19):

$$\forall f_{\beta\to\gamma}\ xs_{\beta\ \mathsf{list}}\ ys_{\beta\ \mathsf{list}}.\ \mathsf{inj}\ f \longrightarrow (\mathsf{map}\ f\ xs = \mathsf{map}\ f\ ys) \longleftrightarrow (xs = ys) \tag{19}$$

$$\forall f_{\alpha\to\gamma}.\ \forall xs\ ys \in \mathsf{lists}\ A_{\alpha\ \mathsf{set}}.\\ \quad \mathsf{inj}_{\mathsf{on}}\ A\ f \longrightarrow (\mathsf{map}\ f\ xs = \mathsf{map}\ f\ ys) \leftrightarrow (xs = ys) \tag{20}$$

First of all, we reformulate the problem a little bit. We will not talk about isomorphisms Abs and Rep but express the isomorphism between A and β by a binary relation $\mathsf{T}_{\alpha\to\beta\to\mathsf{bool}}$ defined as $\mathsf{T}\ x\ y = (\mathsf{Rep}\ y = x)$. We call T a transfer relation. Notice that T captures inversion of injection of β into α and therefore the relation is right-total ($\forall y.\ \exists x.\ R\ x\ y$), right-unique ($\forall x\ y\ z.\ R\ x\ y \longrightarrow R\ x\ z \longrightarrow y = z$) and left-unique ($\forall x\ y\ z.\ R\ x\ z \longrightarrow R\ y\ z \longrightarrow x = y$).

The Transfer tool requires two sorts of setup as input: 1) we need more structural information about the type constructors in φ (more than that they are just sets of elements); 2) we need to know how to transfer the constants in φ along the transfer relation T.

Concerning 1), we assume that there exists a relator for every nonnullary type constructor in φ. Relators lift relations over type constructors: Related data structures have the same shape, with pointwise-related elements (e.g., the relator list_all2 for lists), and related functions map related input to related output. Concrete definitions follow:

$$\mathsf{list\_all2} : (\alpha \to \beta \to \mathsf{bool}) \to \alpha\ \mathsf{list} \to \beta\ \mathsf{list} \to \mathsf{bool}$$
$$(\mathsf{list\_all2}\ R)\ xs\ ys \equiv (\mathsf{length}\ xs = \mathsf{length}\ ys) \wedge (\forall (x, y) \in \mathsf{set}\ (\mathsf{zip}\ xs\ ys).\ R\ x\ y)$$
$$\mathrel{\Rightarrow} : (\alpha \to \gamma \to \mathsf{bool}) \to (\beta \to \delta \to \mathsf{bool}) \to (\alpha \to \beta) \to (\gamma \to \delta) \to \mathsf{bool}$$
$$(R \mathrel{\Rightarrow} S)\ f\ g \equiv \forall x\ y.\ R\ x\ y \longrightarrow S\ (f\ x)\ (g\ y)$$

Concerning 2), we need a transfer rule for every constant present in φ. The transfer rules express how constants on α relate to constants on β along T. Let us look at some examples:

$$((\mathsf{T} \mathrel{\Rightarrow} =) \mathrel{\Rightarrow} =)\ (\mathsf{inj}_{\mathsf{on}}\ A)\ \mathsf{inj} \tag{21}$$
$$((\mathsf{T} \mathrel{\Rightarrow} =) \mathrel{\Rightarrow} =)\ (\forall\_ \in A)\ (\forall) \tag{22}$$
$$((\mathsf{list\_all2}\ \mathsf{T} \mathrel{\Rightarrow} =) \mathrel{\Rightarrow} =)\ (\forall\_ \in \mathsf{lists}\ A)\ (\forall) \tag{23}$$
$$((\mathsf{T} \mathrel{\Rightarrow} =) \mathrel{\Rightarrow} \mathsf{list\_all2}\ \mathsf{T} \mathrel{\Rightarrow} \mathsf{list\_all2} =)\ \mathsf{map}\ \mathsf{map} \tag{24}$$
$$(\mathsf{list\_all2}\ \mathsf{T} \mathrel{\Rightarrow} \mathsf{list\_all2}\ \mathsf{T} \mathrel{\Rightarrow} =)\ (=)\ (=) \tag{25}$$

As already mentioned, the universal quantification on β corresponds to a bounded quantification over A on α ($\forall\_ \in A$). The relation between the two constants is obtained purely syntactically from the type of the constant on β (e.g., $(\beta \to \gamma) \to \mathsf{bool}$ for inj): We replace every type that does not change (γ and bool) by the identity relation $=$, every nonnullary type constructor by its corresponding relator ($\to$ by $\mathrel{\Rightarrow}$ and list by list_all2) and every type that changes by the corresponding transfer relation (β by T).

Having the appropriate setup, the Transfer tool builds a derivation tree whose root is the equivalence theorem between (19) and (20), whose leaves contain the above-stated transfer rules (21)–(25) and whose other nodes are derived by the following three structural rules (for a bound variable, application and lambda abstraction):

$$\frac{R\ x\ y \in \Gamma}{\Gamma \vdash R\ x\ y} \qquad \frac{\Gamma_1 \vdash (R \mathrel{\Rightarrow} S)\ f\ g \qquad \Gamma_2 \vdash R\ x\ y}{\Gamma_1 \cup \Gamma_2 \vdash S\ (f\ x)\ (g\ y)} \qquad \frac{\Gamma,\ R\ x\ y \vdash S\ (f\ x)\ (g\ y)}{\Gamma \vdash (R \mathrel{\Rightarrow} S)\ (\lambda x.\ f\ x)\ (\lambda y.\ g\ y)}$$

Similarity of the rules to those for typing of the simply typed lambda calculus is not a coincidence. A typing judgment here involves two terms instead of one, and a binary relation takes the place of a type. The environment Γ collects the local assumptions for bound variables. Thus since (19) and (20) are of type bool, the procedure produces (19) = (20) as the corresponding relation for bool is $=$. Overall, if the Transfer tool has appropriate setup

available (for type constructors and transfer rules), it can automatically derive the equivalence theorem for any closed lambda term.

Finally, let us discuss for which class of formulas we can obtain the appropriate setup. First, the setup for type constructors:[8] Besides the manual setup for the function type (it is a part of Isabelle's library), the Transfer tool generates the setup automatically for every type constructor that is a natural functor (sets, finite sets, all algebraic datatypes and codatatypes) [39].

Second, transfer rules: Instead of providing a specific transfer rule for every particular transfer relation (such as T) and for every type instance of the given constant (for $\forall$ we needed two rules (22) and (23)), the Transfer tool needs only one general rule for a given polymorphic constant—the specific ones are derived internally from the general one. These general rules capture a certain notion of restricted parametricity[9] and take a transfer relation as a parameter. Since we transfer only over relations that are inversion of injections in our Types-to-Sets project, we can additionally assume that the relation is right-total, right-unique and left-unique. Such parametricity rules hold for $\forall$ and $=$:

$$\mathsf{right\_total}\ R \longrightarrow ((R \Rightarrow\!\!\!\vDash =) \Rightarrow\!\!\!\vDash =)\ (\forall\_ \in (\mathsf{Domain}\ R))\ (\forall)$$
$$\mathsf{left\_unique}\ R \longrightarrow \mathsf{right\_unique}\ R \longrightarrow (R \Rightarrow\!\!\!\vDash R \Rightarrow\!\!\!\vDash =)\ (=)\ (=)$$

Such parametricity rules hold for a broad class of functions—in particular, transitively for any function whose defining term is build from equality and logical quantifiers. The only troublemaker here is the choice operator, which is in general not parametric.

To conclude, the Transfer tool can automatically produce relativization of any theorem that contains only 1) types built from the function type and natural functors and 2) constants that are parametric w.r.t. right-total, right-unique and left-unique relations.

8 Beyond Types to Sets: Types to Terms

So far we have observed that types are semantically equivalent to nonempty sets in HOL and developed a technique that allows us to translate types into sets in a principled way. In this section, we go beyond the connection types–sets and look at how types can represent mathematical objects different from sets (for example, numbers) and thus simulate restricted dependent types in HOL. We show how to use our Types-to-Sets mechanism to compile out such pseudo-dependent types back into terms in case they become too restrictive.

8.1 Harrison's Trick

HOL does not support dependent types. For example, one cannot define a type of vectors of length n by a type definition that takes the term n as a parameter. However, making use of polymorphism, the type definition could be parametrized by a type whose cardinality

[8] The setup requires more than just the name of a relator. Besides making sure that the relator satisfies many natural properties such as monotonicity or compositionality, we use other concepts such as the knowledge that "lists whose elements are in A" can be expressed by $\mathsf{lists}\ A$. See the complete description of the required structure in the first author's thesis [30, §4.7].

[9] These rules are related to Reynolds's relational parametricity [37] and Wadler's free theorems [40]. The Transfer tool is a working implementation of Mitchell's representation independence [32] and it demonstrates that transferring of properties across related types can be organized and largely automated using relational parametricity.

would *encode* the length n. This technique is known under the appellation *Harrison's trick* in the HOL community—it was introduced in John Harrison's work on Euclidean spaces [20], where the spaces $\mathbb{R}^n$ are represented by a type constructor α rvec, with the cardinality of α encoding the dimension n [19, §1-2].

This allows us to prove theorems about Euclidean spaces of any dimension and obtain theorems about specific sizes by taking suitable type instances. For example, a theorem about $\mathbb{R}^3$ is obtained by instantiating α in α rvec to any type of cardinality 3. Harrison also used the trick to encode compatibility of dimensions in matrix multiplication—making the dimensions automatically inferable (most of the times) by HOL's type inference. Naturally, this trick is still far from the expressiveness of full-fledged dependent types; in particular, one cannot perform arithmetic operations or induction on these type variables (at least not without a complex automation setup).

In Isabelle/HOL, we can additionally use the type classes (recall they are essentially predicates on types) to further improve the encoding: For example, we would use α_{finite} rvec to avoid any special treatment for infinite types α, whereas these are artificially designated to represent dimension 1 in Harrison's definition.

8.2 Finite Fields as a Type

Divasón et al. used Harrison's trick in their recent work [14] to encode Galois fields of prime order $\mathsf{GF}(p)$, which are finite fields of prime order (i.e., prime cardinality). These are represented as the type α_{pc} mod-ring, where:

- pc is the class of finite types of prime cardinality—note that pc is a subclass of finite
- mod-ring is a type constructor defined as α_{finite} mod-ring $= \{0, \ldots, |\alpha| - 1\}$, i.e., consisting of the first $|\alpha|$ numbers

We will abbreviate α_{pc} mod-ring as α_{pc} GF.

Besides the usual advantage of types making proofs easier, defining prime fields as a (polymorphic) type enables the reuse of type-based results from Isabelle's library (such as Gauss-Jordan elimination working over any α_{field}).

While using the type α_{pc} GF, Divasón et al. faced a natural limitation of Harrison's trick. They work with an (executable) factorization algorithm factor : α_{pc} GF poly $\to \alpha_{\mathsf{pc}}$ GF $\times$ α_{pc} GF poly, where β poly is the type of polynomials over β, with β assumed to be a ring (as usual, via the type class mechanism). As long as the field's order is static (e.g., when one executes factor directly), one can manually obtain the appropriate instance of factor. But a problem arises if factor is used as a black box in a bigger algorithm in which the order is computed dynamically. The algorithm would have to dynamically create a type of the given cardinality to obtain the right instance of factor—this is not possible in HOL.

Divasón et al. solved the problem by using Local Typedef to compile out the implicit type parameter into an explicit term parameter in two steps. First, they defined factor's term-based version, factor$'$: nat $\to$ int poly $\to$ int $\times$ int poly, where the type int (of integers) is the "universe" holding the carriers of all finite fields. The order p of the field is now an explicit term parameter, as in factor$'$ p. They defined a relation between α GF and int and used the Transfer tool to move between these two representations and to obtain the correctness theorem for factor$'$ from the correctness theorem for factor. Concretely, given the theorem $\varphi[\alpha_{\mathsf{pc}}\ \mathsf{GF}, \mathsf{factor}]$, Transfer produces a theorem

$$\mathsf{prime}\ p \longrightarrow p = |\alpha_{\mathsf{pc}}| \longrightarrow \varphi'[\mathsf{int}, \mathsf{factor}'\ p]$$

where φ' is the isomorphic image of φ along the relation between α GF and int.

Second, since α is not present in φ', the assumption about α really means "for any given prime p there exists a type of cardinality p." Local Typedef allowed them to remove this assumption after proving "for any given prime p, there exists a nonempty subset of natural numbers such that the subset has cardinality p" (which is trivial, with witness $\{0, \ldots, p-1\}$). Divasón et al. have automated the second step for their specific use case.

8.3 Types to Terms: General Case

We now describe how to carry out the second step of Divasón et al.'s approach in general. Recall that our goal is to use types to implicitly encode some "term information" in order to overcome the lack of dependent types. This can be achieved through a decoding function $f : \forall \alpha\gamma.\ \sigma$, where the type σ does not contain α.[10] Thus any type τ of class Υ represents the term "f applied to τ," of type σ.

When working with types becomes too restrictive (e.g., we want to make induction over $\alpha\gamma$), we need to switch back to terms (which allows us take advantage of existing induction principles for σ), and transfer affected type-based theorems to term-based ones. Say we have proved $\forall \alpha\gamma.\ \varphi[\alpha]$, where $\varphi[\alpha]$ is a polymorphic formula that represents a property about the encoded (term) entities. From this, applying the decoding function together with ad hoc knowledge about the behavior of the encoding, we obtain a term-based version of the above theorem, $\forall \alpha\gamma.\ \varphi'[f(\alpha)]$.

However, this is not good enough, since the theorem still depends on α and requires f to be applied explicitly to α. Let us imagine for a moment that f were an ordinary function (from terms to terms), say, $f : \tau \to \sigma$. Then we could easily remove f from $\forall x_\tau.\ \varphi'[f\ x]$, obtaining $\forall y_\sigma \in \mathsf{range}\ f.\ \varphi'[y]$ (where $\mathsf{range}\ f$ can be described independently of f in most practical applications). The idea is to use Types-to-Sets to translate the "types-to-terms" f into a related "terms-to-terms" function as imagined above.

We assume that there exists f's relativization $f^{\mathsf{on}} : \alpha\ \mathsf{set} \to \sigma$ together with the corresponding transfer rule. Applying our Types-to-Sets tool from Section 6 to the formula $\forall \alpha\gamma.\ \varphi'[f(\alpha)]$, we obtain its relativization to sets

$$\forall \alpha.\ \forall A_{\alpha\ \mathsf{set}}.\ A \neq \emptyset \longrightarrow \Upsilon^{\mathsf{on}}\ A \longrightarrow \varphi'[f^{\mathsf{on}}\ A].^{11}$$

Finally, let us assume there exist[12] representation type τ and $g : \sigma \to \tau\ \mathsf{set}$ that is a right inverse for τ-instance of f^{on}, namely $f^{\mathsf{on}} : \tau\ \mathsf{set} \to \sigma$, on some $S_{\sigma\ \mathsf{set}}$. Then we can compile f^{on} out by rewriting the last theorem into the desired form

$$\forall y_\sigma \in S.\ g\ y \neq \emptyset \longrightarrow \Upsilon^{\mathsf{on}}\ (g\ y) \longrightarrow \varphi'[y].$$

For the Galois field example, we use the following instantiation: $\sigma = \mathsf{nat}$, $f(\alpha) : \mathsf{nat} = |\alpha|$, $f^{\mathsf{on}} : \alpha\ \mathsf{set} \to \mathsf{nat} = |.|$, $S = \mathbb{N}$, $g : \mathsf{nat} \to \mathsf{nat\ set} = \lambda p.\ \{0, \ldots, p-1\}$ and $\Upsilon^{\mathsf{on}} : \alpha\ \mathsf{set} \to \mathsf{bool} = \lambda A.\ \mathsf{prime}\ |A|$. Since it holds that $\Upsilon^{\mathsf{on}}\ (g\ p) \longleftrightarrow \mathsf{prime}\ p$ and $g\ p \neq \emptyset$ for every prime number p, we obtain

$$\forall p_{\mathsf{nat}}.\ \mathsf{prime}\ p \longrightarrow \varphi'[\mathsf{int}, \mathsf{factor}'\ p].$$

[10] The type $\forall \alpha\gamma.\ \sigma$ is not directly expressible in HOL but we can use Wenzel's trick and write $\alpha\gamma\ \mathsf{itself} \to \sigma$; see footnote 5 on page 13.

[11] We assumed that the type class Υ does not have any associated operations. Lifting the description to the most general version of Υ is analogous to the approach in Section 6.4 and we omit it here.

[12] In the worst case, we can always set S to be the range of f^{on} and define g by choice.

We showed how Types-to-Sets allows us to compile out Harrison's trick when we reach its limits. This matches our introduction discussion on chasing the sweet spot between expressiveness and complexity: We can bring some of the dependent type expressiveness into HOL theorem provers and still offer an exit strategy once the approach turns out to be too restrictive. We plan to look more deeply into these unknown waters to see how far we can reliably and flexibly bend HOL towards dependent types.

9 Limitations and Future Work

We already mentioned that the dictionary construction cannot be performed for a type constructor depending on overloaded constants. To workaround the problem, we could also "unfold" the definition of such a type and relativize the formula to the type's defining predicate as we did in our work on conservativity of type definitions in HOL and Isabelle/HOL [29]. Whether this approach would yield still practically usable theorems remains an open problem.

As mentioned, we implemented the proposed extensions (the (LT) and (UO) rules), which are part of the Isabelle distribution since Isabelle2016-1. Our future work is to implement an infrastructure that would streamline the relativization process as follows: it would automatically define the relativized constants (all the $c^{\mathsf{on}}_{\mathsf{with}}$ constants), prove their transfer rules w.r.t. their nonrelativized counterparts and call the Transfer tool. These steps must be performed manually at the moment. We do not expect any principal problems; it is just an engineering task that must be done.

With the infrastructure in place, we plan to perform a case study, in which we would transform one of the set-based formalizations into a type-based one and obtain the set-based formulation by our Types-to-Sets tool. A natural candidate for such a case study would be Isabelle's HOL-Algebra library [1], which does not use the type classes system and instead represents various algebraic structures by locales, which are user space parametrized theory modules with assumptions [7]. Every such a locale represents the algebra's axioms by assumptions and is parametrized by the algebra's explicit carrier and its operations. This is a truly set-based approach.

The HOL-Algebra example exposes another limitation of Isabelle's type system: type classes can have only one type parameter. Since we translate sets into type parameters, we cannot express algebraic structures with multiple set parameters (carriers) as a type class. Indeed, our case study would have to stop at modules, which are parametrized by two sets (corresponding to a "scalar" ring and a "vector" abelian group). In short, not every locale can be transformed into a corresponding type class.

Since all the derived type definitional commands in the various HOL systems are implemented in terms of the foundational type definition principle and this paper shows how to localize this principle, we could in principle localize all the derived definitional commands: we could have local quotients, local datatypes or local records. Needless to say, the local variants would be restricted to only monomorphic definitions as the Local Typedef is. Whether this localization proposal is worth the effort, only time will tell.

10 Conclusion

In this paper, we proposed extending Higher-Order Logic with a Local Typedef (LT) rule. We showed that the rule is not an ad hoc, but a natural addition to HOL in that it incarnates a semantic perspective characteristic to HOL: for every nonempty set A, there must be a type

that is isomorphic to A. At the same time, (LT) is careful not to introduce dependent types since it is an open question how to integrate them into HOL.

We demonstrated how the rule allows for more flexibility in the proof development: with (LT) in place, the HOL users can enjoy succinctness and proof automation provided by types during proving, while still having access to the more widely applicable, set-based theorems.

Being natural, semantically well justified and useful, we believe that the Local Typedef rule is a good candidate for HOL citizenship. We have implemented this extension in Isabelle/HOL, but its implementation should be straightforward and noninvasive in any HOL prover. And in a more expressive prover, such as HOL-Omega [22], this rule could simply be added as an axiom in the user space.

In addition, we showed that our method for relativizing theorems is applicable to types restricted by type classes as well, if we extend the logic by a rule for compiling out overloading constants (UO). With (UO) in place, the Isabelle users can reason abstractly using type classes, while at the same time having access to different instances of the relativized result.

All along according to the motto: *Prove easily and still be flexible.*

Acknowledgments We thank reviewers for useful comments and suggestions. The ITP 2016 reviewers helped us to improve the previous conference version of the paper. We thank Fabian Immler and Dmitriy Traytel for interesting discussions on Types to Terms and HOL dependent typing, respectively. We are indebted to Johannes Hölzl to introduce us to HOL-Algebra and to remind us that not every locale can be translated into a corresponding type class in Isabelle. We gratefully acknowledge support from DFG through grant NI 491/13-3 and from EPSRC through grant EP/N019547/1.

References

1. The HOL-Algebra Library. URL `http://isabelle.in.tum.de/library/HOL/HOL-Algebra/`
2. The HOL4 Theorem Prover. URL `http://hol.sourceforge.net/`
3. Types to Sets in the Isabelle distribution. URL `https://isabelle.in.tum.de/dist/library/HOL/HOL-Types_To_Sets/index.html`
4. Adams, M.: Introducing HOL Zero - (Extended Abstract). In: K. Fukuda, J. van der Hoeven, M. Joswig, N. Takayama (eds.) ICMS 2010, *LNCS*, vol. 6327, pp. 142–143. Springer (2010)
5. Aransay, J., Ballarin, C., Rubio, J.: A Mechanized Proof of the Basic Perturbation Lemma. J. Autom. Reasoning **40**(4), 271–292 (2008)
6. Asperti, A., Ricciotti, W., Coen, C.S., Tassi, E.: The Matita interactive theorem prover. In: CADE-23, pp. 64–69 (2011)
7. Ballarin, C.: Locales: A module system for mathematical theories. J. Autom. Reasoning **52**(2), 123–153 (2014)
8. Bancerek, G., Byliński, C., Grabowski, A., Korniłowicz, A., Matuszewski, R., Naumowicz, A., Pąk, K., Urban, J.: Mizar: State-of-the-art and beyond. In: M. Kerber, J. Carette, C. Kaliszyk, F. Rabe, V. Sorge (eds.) Intelligent Computer Mathematics, pp. 261–279. Springer (2015)
9. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer (2004)
10. Bove, A., Dybjer, P., Norell, U.: A Brief Overview of Agda - A Functional Language with Dependent Types. In: S. Berghofer, T. Nipkow, C. Urban, M. Wenzel (eds.) TPHOLs 2009, *LNCS*, vol. 5674, pp. 73–78. Springer (2009)
11. Chan, H., Norrish, M.: Mechanisation of AKS Algorithm: Part 1 – The Main Theorem. In: C. Urban, X. Zhang (eds.) ITP 2015, *LNCS*, vol. 9236, pp. 117–136. Springer (2015)
12. Coble, A.R.: Formalized Information-Theoretic Proofs of Privacy Using the HOL4 Theorem-Prover. In: N. Borisov, I. Goldberg (eds.) PETS 2008, *LNCS*, vol. 5134, pp. 77–98. Springer (2008)
13. Constable, R.L., Allen, S.F., Bromley, H.M., Cleaveland, W.R., Cremer, J.F., Harper, R.W., Howe, D.J., Knoblock, T.B., Mendler, N.P., Panangaden, P., Sasaki, J.T., Smith, S.F.: Implementing Mathematics with the Nuprl Proof Development System. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1986)

14. Divasón, J., Joosten, S., Thiemann, R., Yamada, A.: A formalization of the Berlekamp-Zassenhaus factorization algorithm. In: CPP, pp. 17–29 (2017)
15. Gordon, M.J.C., Melham, T.F. (eds.): Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. Cambridge University Press (1993)
16. Haftmann, F., Wenzel, M.: Constructive Type Classes in Isabelle. In: T. Altenkirch, C. McBride (eds.) TYPES 2006, *LNCS*, vol. 4502, pp. 160–174. Springer (2006)
17. Harrison, J.: HOL Done Right (1995). URL `http://www.cl.cam.ac.uk/~jrh13/papers/holright.html`
18. Harrison, J.: HOL Light: A Tutorial Introduction. In: M. K. Srivas, A.J. Camilleri (eds.) FMCAD '96, *LNCS*, vol. 1166, pp. 265–269. Springer (1996)
19. Harrison, J.: A HOL Theory of Euclidean space. In: J. Hurd, T. Melham (eds.) TPHOLs 2005, *LNCS*, vol. 3603. Springer-Verlag, Oxford, UK (2005)
20. Harrison, J.: The HOL Light theory of Euclidean space. J. Autom. Reasoning **50**, 173–190 (2013)
21. Hölzl, J., Heller, A.: Three Chapters of Measure Theory in Isabelle/HOL. In: M.C.J.D. van Eekelen, H. Geuvers, J. Schmaltz, F. Wiedijk (eds.) ITP 2011, *LNCS*, vol. 6898, pp. 135–151. Springer (2011)
22. Homeier, P.V.: The HOL-Omega logic. In: S. Berghofer, T. Nipkow, C. Urban, M. Wenzel (eds.) TPHOLs 2009, *LNCS*, vol. 5674, pp. 244–259. Springer (2009)
23. Huffman, B., Kunčar, O.: Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. In: G. Gonthier, M. Norrish (eds.) CPP 2013, *LNCS*, vol. 8307, pp. 131–146. Springer (2013)
24. Immler, F.: Generic Construction of Probability Spaces for Paths of Stochastic Processes. Master's thesis, Institut für Informatik, Technische Universität München (2012)
25. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers (2000)
26. Krauss, A., Schropp, A.: A Mechanized Translation from Higher-Order Logic to Set Theory. In: M. Kaufmann, L.C. Paulson (eds.) ITP 2010, *LNCS*, vol. 6172, pp. 323–338. Springer (2010)
27. Kunčar, O., Popescu, A.: From Types to Sets by Local Type Definition in Higher-Order Logic. In: J.C. Blanchette, S. Merz (eds.) ITP 2016, *LNCS*, vol. 9807, pp. 200–218. Springer (2016)
28. Kunčar, O., Popescu, A.: Comprehending Isabelle/HOL's consistency. In: H. Yang (ed.) ESOP 2017, *LNCS*, vol. 10201, pp. 724–749. Springer (2017)
29. Kunčar, O., Popescu, A.: Safety and conservativity of definitions in HOL and Isabelle/HOL. Proc. ACM Program. Lang. **2**(POPL), 24:1–24:26 (2017)
30. Kunčar, O.: Types, Abstraction and Parametric Polymorphism in Higher-Order Logic. Ph.D. thesis, Fakultät für Informatik, Technische Universität München (2016). URL `http://www21.in.tum.de/~kuncar/documents/kuncar-phdthesis.pdf`
31. Maggesi, M.: A formalisation of metric spaces in HOL Light (2015). URL `http://www.cicm-conference.org/2015/fm4m/FMM_2015_paper_3.pdf`. Presented at the Workshop Formal Mathematics for Mathematicians. CICM 2015. Published online.
32. Mitchell, J.C.: Representation Independence and Data Abstraction. In: POPL '86, pp. 263–276. ACM (1986)
33. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, *LNCS*, vol. 2283. Springer (2002)
34. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Part of the Isabelle2015 distribution (2015). URL `https://isabelle.in.tum.de/dist/Isabelle2015/doc/tutorial.pdf`
35. Nipkow, T., Snelting, G.: Type Classes and Overloading Resolution via Order-Sorted Unification. In: J. Hughes (ed.) Functional Programming Languages and Computer Architecture, *LNCS*, vol. 523, pp. 1–14. Springer (1991)
36. Pitts, A.: Introduction to HOL: A Theorem Proving Environment for Higher Order Logic, chap. The HOL Logic, pp. 191–232. In: Gordon and Melham [15] (1993)
37. Reynolds, J.C.: Types, Abstraction and Parametric Polymorphism. In: IFIP Congress, pp. 513–523 (1983)
38. Shankar, N., Owre, S., Rushby, J.M.: PVS Tutorial. Computer Science Laboratory, SRI International (1993)
39. Traytel, D., Popescu, A., Blanchette, J.C.: Foundational, Compositional (Co)datatypes for Higher-Order Logic: Category Theory Applied to Theorem Proving. In: LICS 2012, pp. 596–605. IEEE (2012)
40. Wadler, P.: Theorems for Free! In: FPCA '89, pp. 347–359. ACM (1989)
41. Wadler, P., Blott, S.: How to Make Ad-hoc Polymorphism Less Ad Hoc. In: POPL '89, pp. 60–76. ACM (1989)
42. Wenzel, M.: Type Classes and Overloading in Higher-Order Logic. In: E.L. Gunter, A.P. Felty (eds.) TPHOLs '97, *LNCS*, vol. 1275, pp. 307–322. Springer (1997)
43. Wickerson, J.: Isabelle Users List (2013). URL `https://lists.cam.ac.uk/mailman/htdig/cl-isabelle-users/2013-February/msg00222.html`