From Types to Sets in Isabelle/HOL
Extended Abstract

Ondřej Kunčar[1] and Andrei Popescu[1,2]

[1] Fakultät für Informatik, Technische Universität München, Germany
[2] Institute of Mathematics Simion Stoilow of the Romanian Academy, Bucharest, Romania

Abstract. HOL types 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 analyze some undesired consequences of this limitation and propose a more expressive type-definition rule that addresses it. The new expressive power opens the opportunity for a package that relativizes type-based statements to more flexible set-based variants—to streamline this process, we further implement a rule that transforms the implicit type-class constraints into explicit assumptions. Moreover, our tool is an interesting use case of Lifting and Transfer.

1 Motivation

Recall that the Isabelle/HOL constant $\mathsf{lists} : \alpha\ \mathsf{set} \to \alpha\ \mathsf{list}\ \mathsf{set}$ takes a set A and returns the set of lists whose elements are in A. Let $P : \alpha\ \mathsf{list} \to \mathsf{bool}$ be a fixed constant (whose definition is not important here). Consider the following HOL statements, where we indicate explicitly the top quantification over types:

(1) $\forall\alpha.\ \exists xs : \alpha\ \mathsf{list}.\ P\ xs$
(2) $\forall\alpha.\ \forall A : \alpha\ \mathsf{set}.\ A \neq \emptyset \to (\exists xs \in \mathsf{lists}\ A.\ P\ xs)$

(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 from A. From the set-theoretic semantics point of view, the two statements are obviously equivalent. However, from a theorem proving perspective, they are quite different. While it is much easier to reason about "type-based" statements such as (1), HOL users often need more flexible "set-based" statements such as (2).

Ideally, the users would develop their theories in a type-based fashion, and then export the main theorems as set-based statements. Unfortunately, the HOL systems currently do not allow for this—for example, assuming that (1) is a theorem, one cannot prove (2)! Indeed, in a proof attempt of (2), one would fix a nonempty A and, to invoke (1), one would need to define a new type corresponding to A, an action not currently allowed inside a proof context.

This problem bites even stronger when it comes to package writing: for example, the new (co)datatype package maintains a notion of bounded natural functor, which

in the unary case is a type constructor $\alpha\ \mathsf{F}$ together with a functorial map function $\mathsf{Fmap} : (\alpha \to \beta) \to \alpha\ \mathsf{F} \to \beta\ \mathsf{F}$. For technical reasons, some key facts proved by the package (e.g., those involving algebras and coalgebras for F) require the more flexible set-based variant—this is done via an internalization of F to sets, $\mathsf{Fin} : \alpha\ \mathsf{set} \to \alpha\ \mathsf{F}\ \mathsf{set}$.[3] The development would be dramatically simplified if one could focus on the type-based counterparts for the intermediate lemmas, and only export the set-based version of the main results at the end.

2 Proposal of a Logic Extension

To address the above, we propose extending the HOL logic with a new rule for type definition having the following properties:

- It enables type definitions inside proofs while avoiding the introduction of dependent types by a simple syntactic check
- It is natural and sound w.r.t. the HOL standard model

Recall that the current Isabelle/HOL type definition mechanism[4] employs the constant $\mathsf{typedef} : (\alpha \to \beta) \to (\beta \to \alpha) \to \alpha\ \mathsf{set} \to \mathsf{bool}$, with $\mathsf{typedef}_{\alpha,\beta}\ Abs\ Rep\ A$ stating that Abs and Rep are forth and back bijections between the A subset of α and β. When the user issues a command "typedef $\tau = S$" where $S : \sigma\ \mathsf{set}$ is a given subset of a given type σ, the system introduces a new type τ and two constants $Abs_\tau : \sigma \to \tau$ and $Rep_\tau : \tau \to \sigma$ and adds the following axiom: $S \neq \emptyset \to \mathsf{typedef}_{\sigma,\tau}\ Abs\ Rep\ S$. The user is required to discharge the goal $S \neq \emptyset$, after which the theorem $\mathsf{typedef}_{\sigma,\tau}\ Abs_\tau\ Rep_\tau\ S$ is inferred.

Taking a purely semantic perspective and for a minute ignoring the rank-1 polymorphism restriction of HOL, 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:

$$\forall \alpha.\ \forall A : \alpha\ \mathsf{set}.\ A \neq \emptyset \to (\exists \beta\ Abs\ Rep.\ \mathsf{typedef}_{\alpha,\beta}\ Abs\ Rep\ A) \quad (*)$$

The "typedef" mechanism can be regarded as the result of applying a sequence of standard rules for connectives and quantifiers in a more expressive logic, with $(*)$ as an eigenformula:

(1) $\forall$ elimination of α and A with given type σ and term $S : \sigma$ (both provided by the user), and implication elimination:

$$\frac{\Gamma \vdash S \neq \emptyset \qquad \Gamma, \exists \beta\ Abs\ Rep.\ \mathsf{typedef}_{\sigma,\beta}\ Abs\ Rep\ S \vdash \varphi}{\Gamma \vdash \varphi}\ \forall_E, \forall_E, \to_E$$

(2) $\exists$ "left" rule[5] for β, Abs and Rep, introducing some new/fresh τ, Abs_τ and Rep_τ:

[3] Fin is to F what lists is to list.
[4] We restrict the discussion to non-polymorphic types.
[5] In Gentzen system jargon.

$$\dfrac{\Gamma \vdash S \neq \emptyset \qquad \dfrac{\Gamma,\ \mathsf{typedef}_{\sigma,\tau}\ Abs_\tau\ Rep_\tau\ S \vdash \varphi}{\Gamma,\ \exists\beta\ Abs\ Rep.\ \mathsf{typedef}_{\sigma,\beta}\ Abs\ Rep\ S \vdash \varphi}\ \exists_{left}, \exists_{left}, \exists_{left}}{\Gamma \vdash \varphi}\ \forall_E, \forall_E, \to_E$$

The user further discharges $\Gamma \vdash S \neq \emptyset$, and therefore the overall effect of this chain is the sound addition of $\mathsf{typedef}_{\sigma,\tau}\ Abs_\tau\ Rep_\tau\ S$ as an extra assumption when trying to prove an arbitrary fact φ—this shows that adding $\mathsf{typedef}_{\sigma,\tau}\ Abs_\tau\ Rep_\tau\ S$ is conservative, and therefore justifies this new "definition".

What we propose is to use a variant of the above (with fewer instantiations) as an actual rule:

- In step (1) we do not ask the user to provide concrete σ and S, but work with type and term variables α and $A : \alpha\ \mathsf{set}$.
- In step (2), we only apply the left $\exists$ rule to the type β

We obtain:

$$\dfrac{\Gamma \vdash A \neq \emptyset \qquad \dfrac{\Gamma,\ \exists Abs\ Rep.\ \mathsf{typedef}_{\alpha,\beta}\ Abs\ Rep\ A \vdash \varphi}{\Gamma,\ \exists\beta\ Abs\ Rep.\ \mathsf{typedef}_{\alpha,\beta}\ Abs\ Rep\ A \vdash \varphi}\ [\beta\ \text{fresh}]\ \ \exists_{left}}{\Gamma \vdash \varphi}\ \forall_E, \forall_E, \to_E$$

In a notation closer to Isabelle, the overall rule, written (LT) as in "Local Typedef", looks as follows:

$$\dfrac{\Gamma \vdash A \neq \emptyset \qquad \Gamma \vdash (\exists Abs\ Rep.\ \mathsf{typedef}_{\alpha,\beta}\ Abs\ Rep\ A) \Longrightarrow \varphi}{\Gamma \vdash \varphi}\ [\beta\ \text{fresh for}\ \varphi, \Gamma] \quad \text{(LT)}$$

The above discussion shows why (LT) is morally correct—in fact, just like the existing typedef rule, (LT) is sound for the standard set-theoretic semantics of HOL.

3 Applications and Further Extensions

The rule (LT) allows us to handle our motivating examples from Section 1. 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 $\exists Abs\ Rep.\ \mathsf{typedef}_{\alpha,\beta}\ Abs\ Rep\ A$, from which we obtain Abs and Rep such that $\mathsf{typedef}_{\alpha,\beta}\ Abs\ Rep\ A$. From this, (1) with α instantiated to β, and the definition of lists, we obtain $\exists xs \in \mathsf{lists}\ (\mathsf{UNIV} : \beta\ \mathsf{set}).\ P\ xs$. Furthermore, using that Abs and Rep are isomorphisms between $\mathsf{UNIV} : \beta$ and A, we obtain $\exists xs \in \mathsf{lists}\ A.\ P\ xs$, as desired.

Of course, the above relativization process would only be really useful if automated. This is a perfect application for Isabelle's Lifting and Transfer packages [2], which can both automatically synthesize the relativized statement (e.g., (2) from above) and prove it from the original statement (e.g., (1) from above) along the emerging isomorphisms.

There is an Isabelle-specific complication though in the "isomorphic" journey between types and sets: the implicit assumptions on types given by the type classes. For example, let us modify (1) to speak about types of class "finite":

(1') $\forall\alpha : \mathsf{finite}.\ \exists xs : \alpha\ \mathsf{list}.\ P\ xs$

In order to relativize (1'), we first need a version of it which internalizes the type class assumptions:

(1") $\forall\alpha.\ \mathsf{finite}(\alpha) \to (\exists xs : \alpha\ \mathsf{list}.\ P\ xs)$

Again, while morally equivalent to (1'), the statement (1") cannot be proved from (1') in Isabelle/HOL. To cope with this, we propose a new rule, named (TCI) as in "type classes internalization".

Let us introduce some notation to formulate the rule. We assume that the system maintains the following dependency relation. Whenever a new item (constant or type) p is defined (or declared) in terms of another item p', one stores the dependency of p from p'. We write $p \rightsquigarrow p'$ to mean that that p depends on p' (directly or indirectly, via transitivity and/or substitution).

Given two types σ and τ, we write $\sigma \leq \tau$ to mean that σ is an instance of τ. We extend $\leq$ to relate types with sets of types, i.e., $\sigma \leq T$ iff there exists $\tau \in T$ such that $\sigma \leq \tau$. Let C be the set of all constants and, for each $c \in C$, let Δ_c be the finite set of types σ for which c_σ already has an overloaded definition.

Given p being a constant or a type and φ a formula, p is said to *appear in* φ, written $p \Subset \varphi$ if p is a constant instance occurring in φ or a subexpression (subtype) of a type occurring in φ. Now we can formulate our rule:

$$\frac{\varphi}{\forall\overline{c}.\ \mathsf{Ax}_s^{\mathsf{rel}}(\beta,\overline{c}) \Rightarrow \varphi[\beta/\beta :: s, \overline{c}/\overline{c_s}]} \begin{bmatrix} p \not\rightsquigarrow \overline{c_s} \text{ for any } p \Subset \varphi \\ \beta \not\leq \Delta_c \text{ for any } c \in \overline{c_s} \end{bmatrix} \quad \text{(TCI)}$$

The assumption of the rule corresponds to (1')—it is a formula φ that depends on a type variable β of sort s and contains some of the constants $\overline{c_s}$ associated to s.[6] The conclusion corresponds to (1")—it quantifies universally over arbitrary variables $\overline{c}$ having the same types as $\overline{c_s}$ but free of the sort constraint s, assumes for $\overline{c}$ the type class axioms $\mathsf{Ax}_s^{\mathsf{rel}}(\beta,\overline{c})$, and concludes the modification of φ with the s-sort constraints removed and with $\overline{c}$ substituted for $\overline{c_s}$.

We don't introduce (TCI) as a new inference primitive in our system because of its complexity. Instead, we introduce two simpler rules that together give us the rule (TCI). This approach also coincides with the way Haskell-like type classes are implemented in Isabelle. The implementation uses (a) a more primitive version of type classes—axiomatic type classes—that a priori do not know about any operations and (b) a quite flexible mechanism for overloading.

The sort internalization rule Our proposed rule follows Wenzel's approach [3] to representing type classes by internalizing them as predicates on types,[7] i.e., constants of type $\forall\alpha.\ \mathsf{bool}$. (Since this particular type is not allowed in Isabelle, Wenzel uses instead $\alpha\ \mathsf{itself} \to \mathsf{bool}$, where $\alpha\ \mathsf{itself}$ is a singleton type.) The rule simply allows one to infer the "internalized" formula $\mathsf{Ax}_s(\beta) \Rightarrow \varphi[\beta/\beta :: s]$ (which takes the type-class axioms as an assumption) from the original formula φ (which has its types constrained with sorts):

[6] A sort is an intersection of type classes.

[7] Our $\mathsf{Ax}_s(\tau)$ from below coincides with the $\langle\tau : s\rangle$ predicate from [3].

$$\frac{\varphi}{\mathsf{Ax}_s(\beta) \Rightarrow \varphi[\beta/\beta :: s]} \quad \text{(SI)}$$

Note that the converse of this rule already follows from the existing rules in Isabelle.

Conjecture 1. The sort internalization rule (SI) is conservative.
Proof idea. Follows from the semantics of sorts: the sort constrain $\tau :: s$ has the same truth value as $\mathsf{Ax}_s(\tau)$. (See also [3], where $\mathsf{Ax}_s(\tau)$ is explicitly used to encode $\tau :: s$.) □

The unoverloading rule This rule allows us to replace, in any formula φ, a constant instance c_σ by a fresh term variable x_σ provided that (a) c_σ has not been overloaded yet and (b) φ contains no item p that depends on c_σ.

$$\frac{\varphi}{\forall x_\sigma.\, \varphi[x_\sigma/c_\sigma]}\ [p \not\rightsquigarrow c_\sigma \text{ for any } p \in\!\!\in \varphi;\ \sigma \not\leq \Delta_c] \quad \text{(UO)}$$

Conjecture 2. The unoverloading rule (UO) preserves consistency.
Proof idea. Since c_σ is unconstrained (hence uninterpreted), it behaves like a term variable x_σ. Moreover, the truth of a formula φ containing c_σ should not be affected by any existing definition of c_τ with $\tau < \sigma$. □

A realization of (TCI) We achieve the effect of (TCI) by the following sequence of steps:

1. We apply the (SI) rule.
2. We replace $\mathsf{Ax}_s(\alpha)$ by its equivalent relativized version $\mathsf{Ax}_s^{\mathsf{rel}}(\alpha, \overline{c_s})$.
3. If s defines n operations $\overline{c_s}$, we apply the rule (OU) n times, once for each c in $\overline{c_s}$.

With the new rules (LT) and (TCI) in place, the Lifting and Transfer machinery can in principle be used to relativize arbitrary formulas.

4 Conclusion, Ongoing and Future Work

We are currently experimenting with a prototype implementation of the new rules and their integration with the Lifting and Transfer packages—the implementation of the rules Local Typedef and Sort Internalization is already available from [1]. We are also working at a rigorous proof of the rules' soundness, within a set-theoretic semantics of Isabelle/HOL that also explains overloading. We believe that these rules, due to their potential usefulness in serving the users, are good candidates for HOL citizenship.

Acknowledgment. We would like to thank Tobias Nipkow and Makarius Wenzel for many encouraging and controversial discussions about this topic.

References

1. `http://www21.in.tum.de/~kuncar/documents/new_rules.thy`
2. Huffman, B., Kunčar, O.: Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. In: Certified Programs and Proofs. Springer (2013)
3. Wenzel, M.: Type Classes and Overloading in Higher-Order Logic. In: Gunter, E.L., Felty, A.P. (eds.) TPHOLs. Lecture Notes in Computer Science, Springer (1997)