Isomorphic Transfer Infrastructure for Nested Types in Isabelle/HOL (Work in Progress)

Gergely Buday[1], Andrei Popescu[1]

[1]School of Computer Science, University of Sheffield, Sheffield, UK

Abstract

This paper addresses (part of) the problem of simplifying reasoning with proof assistants by transferring theorems that are stated in a heavy form, using explicit invariants, to lightweight counterparts where the invariants are handled implicitly by the type system. Specifically, we provide some abstract assumptions that allow one to establish isomorphisms for nested applications of defined types in Gordon's Higher-Order Logic (HOL). This allows the seamless isomorphic transfer of results across type definitions in the presence of nesting. Our results have been formalized in the Isabelle/HOL theorem prover, and we plan to integrate them with Isabelle's Lifting and Transfer tool.

Keywords

Higher-Order Logic (HOL), type definitions, theorem proving, Isabelle/HOL

1. Introduction

The definition of new types by carving out subsets of existing types, known as *typedef*, is a fundamental mechanism in Higher-Order Logic (HOL), a logic that provides the foundation for several interactive theorem provers, including HOL4 [1], HOL Light [2] and Isabelle/HOL [3]. While most of the uses of typedef are hidden under the (automated) definition of (co)inductive datatypes [4, 5, 6, 7] and therefore not directly invoked by the users, there still remains an important scenario where typedef is invoked explicitly.

Namely, say one has a type (perhaps an algebraic datatype) T that does not capture precisely the intended concept (because it contains too many elements), but only via an *invariant* defined in terms of a predicate on T, or equivalently, a set I that has type $T\ set$.[1] Then one defines the more precise type S as a typedef, to consist of exactly the inhabitants of T that belong to the set I—i.e., I is effectively turned into a type.

Let us consider two examples, which will act as our running examples throughout this paper.

Example 1. ***(Distinct Lists)*** Polymorphic lists are introduced as the following datatype, where we use ML-style notation feature by Isabelle/HOL as well as all the HOL-based provers (in particular, α denotes a type variable):

$$\text{datatype}\ \ \alpha\ list = Nil \mid Cons\ \alpha\ (\alpha\ list)$$

For a list xs, we let $length\ xs$ be its length and, given a natural number $i < length$, we let $xs!i$ be the $(i-1)$'th element in xs (so the indexing starts from 0). In some developments, one may be interested in working with nonrepetitive ("distinct") lists, i.e., lists whose elements do not repeat—to this end, one defines the (polymorphic) predicate $distinct : \alpha\ list \to bool$ by $distinct\ xs \equiv \forall i\ j.\ i < j \wedge j < length\ xs \longrightarrow xs!i \neq xs!j$. The new (polymorphic) type $\alpha\ dlist$ of distinct lists is defined as a typedef:

$$\text{typedef}\ \ \alpha\ dlist = \{xs : \alpha\ list \mid distinct\ xs\}$$

This command introduces the new type $\alpha\ dlist$ together with an abstraction-representation pair of (polymorphic) constants[2] $(Rep_{dlist}, Abs_{dlist})$ with $Rep_{dlist} : \alpha\ dlist \to \alpha\ list$ and $Abs_{dlist} : \alpha\ list \to \alpha\ dlist$

PhD Symposium of the 19th International Conference on Integrated Formal Methods (iFM) at the University of Manchester, UK, 12 November 2024.

g.buday@sheffield.ac.uk (G. Buday); a.popescu@sheffield.ac.uk (A. Popescu)

[1]The type $T\ set$ of sets of elements of a type T is a copy of the type $T \to bool$, where set is a type operator.

[2]In HOL, items that have a fixed interpretation, including fixed values such as 0 or defined functions, are called "constants"; they are to be contrasted with "variables", which do not have a fixed interpretation but range over given types.

and the following axioms stating that these functions are mutually inverse bijections between the new type α *dlist* and the subset $\{xs : \alpha\ list \mid distinct\ xs\}$ of the base type α *list*.[3] In Isabelle/HOL, these axioms are actually packed into a single axiom:

$$type\_definition\ Rep_{dlist}\ Abs_{dlist}\ \{xs : \alpha\ list \mid distinct\ xs\}$$

where the (polymorphic) predicate $type\_definition : (\beta \to \alpha) \to (\alpha \to \beta) \to (\alpha\ set) \to bool$ is defined as follows: $type\_definition\ r\ a\ A \equiv (\forall x.\ r\ x \in A) \wedge (\forall y : \beta.\ a\ (r\ y) = y) \wedge (\forall x : \alpha.\ x \in A \longrightarrow r\ (a\ x) = x)$.

Thus, in this example T is α *list*, I is $\{xs : \alpha\ list \mid distinct\ xs\}$, and S is α *dlist*. □

Example 2. ***(Discrete Distributions)*** A (discrete) distribution is a positive function to the real numbers of countable support such that its values sum up to to 1. The polymorphic type α *distrib* of distributions on α is introduced as a typedef having base type $\alpha \to real$ (the type of functions from α to *real*):

$$\text{typedef}\ \ \alpha\ distrib = \{f : \alpha \to real \mid dist\ xs\}$$

where the predicate $dist : (\alpha \to real) \to bool$ is defined by $dist\ f \equiv (\forall x : \alpha.\ f\ x \geq 0) \wedge countable\ \{x : \alpha \mid f\ x \neq 0\} \wedge \sum_{x:\alpha} f\ x = 1$.

As before, the above command introduces a new type α *distrib* , an abstraction-representation pair of constants (Rep_{dlist}, Abs_{dlist}) with ($Rep_{distrib}$, $Abs_{distrib}$) with $Rep_{distrib} : \alpha\ distrib \to \alpha\ list$, and the axiom $type\_definition\ Rep_{distrib}\ Abs_{distrib}\ \{f : \alpha \to real \mid dist\ f\}$ saying that $Abs_{distrib}$ and $Rep_{distrib}$ are mutually inverse bijections between α *distrib* and $\{f : \alpha \to real \mid distrib\ xs\}$.

Thus, in this example T is $\alpha \to real$, I is $\{f : \alpha \to real \mid dist\ xs\}$, and S is α *distrib*. □

In a formal development that follows the above scheme, one usually distinguishes between:

- developing the "internal" mathematical theory, which usually proceeds without defining S, but instead working with T and stating the theorems relativized to I—for example, proving facts of the form $\forall t : T.\ t \in I \longrightarrow \ldots$
- at the end, defining S and "sealing" the library for export by transferring from T to S all the constants and all the main (exportable) facts that have been proved proved relative to I—for example, turning facts of the form $\forall t : T.\ t \in I \longrightarrow \ldots$ into facts of the form $\forall t' : S. \ldots$

The process of "isomorphically" transferring I-relativized constants and results on T to corresponding constants and results on S, while seemingly conceptually straightforward, turns out to be quite subtle in the presence of higher-order constants. It requires infrastructure for lifting relations along type constructors (known as *relators*), which allows the automated proofs of the transferred theorems from the original ones—this is facilitated in Isabelle/HOL by various dedicated tools [8, 9, 10].

In this short work-in-progress paper, we study a fairly common pattern: the isomorphic transfer in the presence of nested type constructors. We start with motivation in terms of a standard construction applied to our running examples (§2), which leads to formulating the wider scope of the problem. We then work out the solution to the problem in an ad hoc manner on the running examples §3. After that, we are ready to describe our main result: some abstract general structure and conditions that enable this pattern, in that they allow constructing a back-and-forth bijection for transfer (§4), and show how it instantiates to our examples. We conclude with related work and future plans, notably the planned integration of our work into Isabelle's Lifting and Transfer package (§5). Our concepts and results apply to Higher-Order Logic, hence are in principle relevant to any HOL-based provers. We have formalized them in the Isabelle/HOL theorem prover, and the formal scripts are publicly available [11].

[3] These are sometimes called an "embedding-projection pair"; here we prefer terminology that is closer to HOL, referring to a representation function (indicating how elements of the new type are represented/implemented in terms of those of the old one) and an opposite abstraction function.

2. The Concrete Problem

Consider the problem of proving that the type constructors (polymorphic types) α *dlist* and α *distrib* constitute monads [12, 13] or at least monad-like structures—which are very useful properties to have for any (data)type, whenever possible.

According to the above pattern, one wishes to first prove the properties for the underlying (representing types) α *list* and $\alpha \to$ *real* relative to the defining predicates *distinct* and *dist*, and then transfer them to the defined types α *dlist* and α *distrib*. (Not only is this good practice, but in some sense it is the only way to proceed, at least initially when "bootstrapping" a theory for the defined types, given that initially our only means to prove a property on the defined type is to trace it back to a property on the underlying type.)

Some of the monadic structure and properties involve nesting the application of type constructors, for example we want to have a map (functorial-action) operator $map_{distrib} : (\alpha \to \beta) \to (\alpha\ distrib \to \beta\ distrib)$ and join (counit) operator $join_{distrib} : (\alpha\ distrib)\ distrib \to \alpha\ distrib$ satisfying (among others) the associativity law $join_{distrib} \circ (map_{distrib}\ join_{distrib}) = join_{distrib} \circ join_{distrib}$.

How to define such structure and prove such properties on the defined types? Let us start with a simple example that does not involve nesting, namely defining the map operator on $(\alpha \to \beta) \to (\alpha\ distrib \to \beta\ distrib)$ and proving that it preserves identities, in that $\forall d : \alpha\ distrib.\ map_{distrib}\ (id : \alpha \to \alpha)\ d = d$ where *id* denotes the identity function. We define $map_{distrib}$ on any $g : \alpha \to \beta$ and $d : \alpha\ distrib$ by $map_{distrib}\ g\ d = Abs_{distrib}\ (\lambda b : \beta.\ \sum_{a \in g^{-1}\ b} Rep_{distrib}\ d\ a)$. Notice that the definition requires a back-and-forth application of the abstraction and representation functions $Abs_{distrib}$ and $Rep_{distrib}$, with some specific manipulation of items of the underlying type $\alpha \to real$. (In this case, the specific manipulation happens to involve taking the sum of a function in $\alpha \to real$ over all the elements of the g-preimage of g, but the exact nature of such manipulations is not important here.) Now, to prove the desired fact, fix $d : \alpha\ distrib$. In order to show $map_{distrib}\ id\ d = d$, by the injectivity of $Rep_{distrib}$ it suffixes to show $Rep_{distrib}\ (map_{distrib}\ id\ d) = Rep_{distrib}\ d$. Using the definition of $map_{distrib}$ and the fact that $Rep_{distrib}$ is left-inverse to $Abs_{distrib}$, the above is equivalent to $\lambda b : \beta.\ \sum_{a \in id^{-1}\ b} Rep_{distrib}\ d\ a = Rep_{distrib}\ d$, i.e., to $\lambda b : \beta.\ \sum_{a=b} Rep_{distrib}\ d\ a = Rep_{distrib}\ d$, which follows from the properties of sums and function extensionality.

We thus have the following pattern: *To define constants on the defined type and prove properties for them, we need to move back and forth via the abstraction-representation pair and use consequences of the associated type_definition axiom.*

But how to do this in the presence of nested defined type constructors (where the abstraction-representation pairs stemming from type definitions no longer work out of the box)? In the next section, we discuss in an ad hoc manner how to tackle the nested isomorphic transfer problem in the presence of nested types for our two running examples. After that, we will introduce an abstract solution, which covers these two cases and many others.

3. The Solution for Two Concrete Instances

Now let us come back to the original more complex problem, of defining $join_{distrib} : (\alpha\ distrib)\ distrib \to \alpha\ distrib$ (in addition to $map_{distrib}$ which we have already defined) and proving the associativity law.

In order to define $join_{distrib}$, which operates on the nested defined type $(\alpha\ distrib)\ distrib$, we need an understanding of how a counterpart of this operator should act on the (nested application of) the underlying type, i.e., on $(\alpha \to real)real$. To be more exact, we don't need to consider the behavior of such a counterpart on all functions $F : (\alpha \to real) \to real$, but seemingly only on functions that act like distributions, i.e., satisfy $distrib\ F$ (i.e., are positive, have countable support and sum to 1). In fact, upon a closer look we see that the functions of interest are not really distributions on the entire type $\alpha \to real$, but on the subset of $\alpha \to real$ that consists of distributions only, i.e., distributions on the set $\{f : \alpha \to real \mid distrib\ f\}$. In other words, we need a relativized version of the predicate *distrib*, let us

denote it by $distOn : \alpha\ set \to \alpha\ distrib \to bool$, defined by

$$distOn\ A\ f \equiv (\forall x : \alpha.\ x \in A \longrightarrow f\ x \geq 0) \wedge countable\ \{x : \alpha \mid x \in A \wedge f\ x \neq 0\} \wedge \sum_{x \in A} f\ x = 1$$

where we have highlighted the difference from the original predicate *dist*—in that the conditions are not applied to the entire type α, but to a parameter subset $A : \alpha\ set$. Note that we can recover the original predicate as $dist = distOn\ UNIV$, where *UNIV* is the "universal" set covering the entire type.

With this relativized predicate at hand, the picture becomes clear: Given a function $F : (\alpha \to real) \to real$ that acts like a distribution on distributions on α, i.e., such that $distOn\ \{f : \alpha \to real \mid dist\ f\}\ F$ holds, we have the formal means to turn it into a "flat" distribution, let us call it *join F*, on $\alpha \to real$, namely by summation (applying of course a well-known mathematical construction): $join\ F\ x \equiv \sum_{f \in \{f \mid dist\ f\}} F\ f\ x$. Now, to define $join_{distrib}$ from *join*, we need to able to move back and forth between $\alpha\ distrib\ distrib$ and and $(\alpha \to real) \to real$, ideally using a similar infrastructure as the abstraction-representation pair ($Abs_{distrib}$, $Rep_{distrib}$) and predicate *distrib* that allowed us to move between $\alpha\ distrib$ and $\alpha \to real$. And indeed, this is possible:

- We define $dist_2 : ((\alpha \to real) \to real) \to bool$ to be $distOn\ \{f : \alpha \to real \mid dist\ f\}$.
- We define $Abs_{distrib,2} : ((\alpha \to real) \to real) \to (\alpha\ distrib)\ distrib$ by $Abs_{distrib,2}\ F = Abs_{distrib}\ (\lambda\ d : \alpha\ distrib.\ F\ (Rep_{distrib}\ d))$.
- We define $Rep_{distrib,2} : (\alpha\ distrib)\ distrib \to ((\alpha \to real) \to real)$ by $Rep_{distrib,2}\ D = \lambda f : \alpha \to real.\ Rep_{distrib}\ D\ (Abs_{distrib}\ f)$.

Note that we defined $distrib_2$ in line with the above analysis, and defined $Abs_{distrib,2}$ and $Rep_{distrib,2}$ with the aim of achieving a bijective correspondence between $(\alpha\ distrib)\ distrib$ and the elements of $(\alpha \to real) \to real$ satisfyig $dist_2$. Therefore, we can prove that $type\_definition\ Abs_{distrib,2}\ Rep_{distrib,2}\ \{D \mid dist_2\ D\}$ holds.

It now remains to prove the associativity of $join_{distrib}$, which we can rephrase as

$$\forall D : ((\alpha\ distrib)\ distrib)\ distrib.\ join_{distrib}\ (map_{distrib}\ join_{distrib}\ D) = join_{distrib}\ (join_{distrib}\ D)\ .$$

This involves further level of nesting of *distrib*; to this end, by essentially iterating one more step the above construction, we obtain $dist_3 : (((\alpha \to real) \to real) \to real) \to bool$, $Abs_{distrib,3} : (((\alpha \to real) \to real) \to real) \to ((\alpha\ distrib)\ distrib)\ distrib$ and $Rep_{distrib,3} : ((\alpha\ distrib)\ distrib)\ distrib \to (((\alpha \to real) \to real) \to real)$ such that $type\_definition\ Abs_{distrib,3}\ Rep_{distrib,3}\ \{D \mid dist_3\ D\}$ holds. Now we can easily prove associativity similarly to how we proceeded in the non-nested case, but using the appropriate back and forth infrastructure in each case, depending on the level of nesting.

A somewhat similar discussion applies to distinct lists, though the details differ:

- We define $distinct_2 : (\alpha\ list)\ list \to bool$ by $distinct_2\ xss \equiv distinct\ xss \wedge (\forall i < length\ xss.\ distinct\ (xss!i))$.
- We define $Abs_{dlist,2} : (\alpha\ list)\ list \to (\alpha\ dlist)\ dlist$ by $Abs_{list,2}\ xss = Abs_{list}\ (map\ Abs_{list}\ xss)$.
- We define $Rep_{dlist,2} : (\alpha\ dlist)\ dlist \to (\alpha\ list)\ list$ by $Rep_{list,2}\ xss = Rep_{list}\ (map\ Rep_{list}\ xss)$.

where *map* is the standard mapping operator for lists. Again, we have that $type\_definition\ Abs_{dlist,2}\ Rep_{dlist,2}\ \{xss \mid distinct_2\ xss\}$ holds.

With the goal of a general solution in mind, let us note some similarities and commonalities of the above two cases. While for distinct lists the definitions of the one-level-up abstraction and representation functions involve entities of the same kind (namely abstractions for abstractions and representations for representations), in the case of distributions the definitions combine the two, for example the definition of the one-level-up abstraction uses outer abstraction together with inner representation. This is a reflection of lists being a covariant functor and function-space-to-reals being a contravariant functor.

The common pattern of the two is, however, the fact that the one-level-up operators employ composition between an (1) operator and (2) the map function for the given functor applied to an operator. This is manifestly clear for distinct lists, for example the one-level-up abstraction $Abs_{dlist,2}$ is the composition of Abs_{dlist} with $map_{list}\ Rep_{dlist}$; for distributions, this is also seen to be the case, if we note that the map function for the contravariant functor $\alpha \to real$ is $\lambda g : \alpha \to \beta.\ \lambda f : \beta \to real.\ g \circ f$.

Another discrepancy between the two cases is the definition of the one-level-up characteristic predicates ($distrib_2$ versus $distinct_2$); as we will see next, we will be able to uniformly capture both cases under a more general set-lifting operator.

4. An Abstract Formulation of the Problem and a General Solution

We formulate the problem abstractly as follows: *How to lift the abstraction-representation properties characteristic of type definitions to nested applications of the defined type constructor?* In technical terms: Given a polymorphic type $\alpha\ T$ and a polymorphic operator $I : (\alpha\ T)\ set$ such that $I \neq \emptyset$ which produce a type definition

$$\text{typedef}\ \ \alpha\ S = \{x : \alpha\ T \mid x \in I\}$$

what structure and properties are in general required for T and I in order to be able to lift the operator I and abstraction-representation pair (Abs_S, Rep_S), for any n, to n-level operator $I_n : (\alpha\ T^n)\ set$ and abstraction-representation pair ($Abs_{S,n}$, $Rep_{S,n}$) with $Abs_{S,n} : \alpha\ T^n \to \alpha\ S^n$ and $Rep_{T,n} : \alpha\ T^n \to \alpha\ S^n$ such that $type\_definition\ Abs_{S,n}\ Rep_{T,n}\ \{x : \alpha\ T \mid I\ x\}$ holds? (Above, $\alpha\ T^n$ denotes the n'th iteration of the type constructor T, in particular $\alpha\ T^1 = \alpha\ T$ and $\alpha\ T^2 = (\alpha\ T)\ T$; and similarly for S.) Indeed, this would allow us to seamlessly apply to the nested case the same back and forth techniques as in the non-nested case.

In the previous section, we have discussed solutions to two instances of this problem. The first instance is representayive of a wide class of situations, namely polymorphic inductive datatypes (which are all covariant functors) with invariants; the second also has some cousins in the formalization and specification literature, for example the defined types topological filters and of mutisets, as well as variations such as discrete subdistributions. In what follows, we introduce a generalization that covers all these cases.

Assumptions. We assume that the underlying type constructor $\alpha\ T$ comes equipped with an operator $Trel : \alpha\ set \to \beta\ set \to (\alpha \to \beta \to bool) \to (\alpha\ T \to \beta\ T \to bool)$ for lifting relations to T that for bijective relations commutes with composition; namely, letting $bijBetw\ A\ B\ R$ express that the relation R is a bijection between A and B:

(A1) $bijBetw\ A\ B\ P$ and $bijBetw\ B\ C\ Q$ implies $Trel\ A\ C\ (P \circ Q) = Trel\ A\ B\ P \circ Trel\ B\ C\ Q$ for all A, B, C, P, Q

Moreover, we assume that there exists an operator $Iset : \alpha\ set \to (\alpha\ T)\ set$ for lifting sets to T, which is an extension of I in that

(A2) $Iset\ UNIV = I$

and the following hold, where $eqOn\ A\ R$ says that the restriction of R to A is the equality on (i.e., the diagonal of) A:

(A3) $bijBetw\ A\ B\ R$ implies $bijBetw\ (Iset\ A)\ (Iset\ B)\ (Trel\ A\ B\ R)$ for all A, B, R
(A4) $bijBetw\ A\ A\ R$ and $eqOn\ A\ R$ implies $eqOn\ (Iset\ A)\ (Trel\ A\ A\ R)$ for all A, R

T together with the operator $Trel$ forms a relator-like structure [14, 15], similar to those that underlie Isabelle/HOL's transfer tool [8] and datatype specification mechanism [6]. However, this concept comes with an explicit indication of the domain and codomain sets and targets bijections between these sets. In particular, (A1) only requires $Trel$ to commute with (relation) composition when restricted to bijective

relations. While we think of *Trel* as being associated to T, we think of *Iset* as being associated to the invariant I (which it generalizes via (A2)). For example, if T is *list*, then it is reasonable to take *Trel* to be the list relator (relating lists position-wise), but we have no reason to commit to *Iset* as being the standard set operator associated to lists (returning the set of all elements of a list)—rather, the choice of *Iset* will depend on what invariant we want to consider on lists. Of course, as the axiom (A2) suggests, the *Iset* parameter of our abstract framework is reminiscent of the relativized version of the predicate *dist* that we employed in the case of distributions.

Trel and *Iset* are connected by the assumptions (A3) and (A4). Thus, (A3) states that *Trel* lifts bijections between two sets to bijections between the *Iset*-liftings of these sets, which roughly means that *Iset* partially acts like a subrelator of (*T*, *Trel*). Finally, (A4) is an *Iset*-relativization of the standard property of relators of preserving equalities—namely, here we say *Trel* preserves partial equalities w.r.t. *Iset*.

Let us see how to instantiate this framework to our running examples. To this end, we first note that having chosen our assumptions in terms of relations rather than functions allows us to capture both covariant and contravariant cases. For the case of the distribution type α *distrib*, we take:

- $\alpha\ T$ to be $\alpha \to real$;
- $Trel\ A\ B\ R\ f\ g$ to be $\Rightarrow_{real}\ (\lambda a, b.\ a \in A \wedge b \in B \wedge R\ a\ b)\ f\ g \wedge (\forall a \notin A.\ f\ a = \bot) \wedge (\forall b \notin B.\ g\ b = \bot)$, where $\bot$ is a (polymorphic) fixed "undefined" element (which is available in HOL on each type via Hilbert choice) and $\Rightarrow_{real}$ is the *real*-instance of the function-space relator, defined by $\Rightarrow_{real}\ P\ f\ g \equiv \forall a, b.\ P\ a\ b \longrightarrow f\ a = g\ b$;
- *Iset* to be given by the relativized form of the *distOn* predicate, namely $Iset\ A \equiv \{f \mid distOn\ A\ f\}$.

For the case of the distinct-list type α *dlist*, we take:

- $\alpha\ T$ to be α *list*;
- $Trel\ A\ B$ to be the list relator *list_all*, where *list_all R* relates two lists just in case they have the same length and their elements are position-wise related (thus, the domain and codomain sets are ignored by *Trel*);
- *Iset* to be defined as $Iset\ A \equiv \{xs \mid distinct\ xs \wedge set\ xs \subseteq A\}$, where $set : \alpha\ list \to \alpha\ set$ is the support operator for lists (returning the set of all elements of a list).

Note the two flavors of instantiating the relativized operator *Iset*, depending on whether we deal with a contravariant or covariant functor, namely : (1) either, in the contravariant case, by relativizing the original predicate *dist* to *distOn*; (2) or, in the covariant case, by intersecting the original predicate *distinct* with the adjoint of the support operator (which is usually available for covariant functors, and in particular is available for all container types)—indeed, the righthand side of the definition of *Iset A* can be written as $\{xs \mid distinct\ xs\} \cap \{xs \mid set\ xs \subseteq A\}$, and the operator $L = \lambda A.\ \{xs \mid set\ xs \subseteq A\}$ is the right adjoint[4] of the support operator *set* in the sense that $xs \in L\ A$ iff $set\ xs \subseteq A$. It is relatively easy to check that these instances satisfy our assumptions. For example, (A3) in the case of the distribution instantiation says that a bijection between two sets A and B induces a bijection the sets of distributions on A and B respectively, which are constant $\bot$ outside A and B respectively.

Our main result is that these abstract assumptions are sufficient for solving our problem, thus generalizing the constructions in the above particular cases (and in many other cases, e.g., any datatypes with invariants turned into typedefs).

Theorem: Under the assumptions (A1)–(A4) above, we have a solution to our problem for all $n \geq 1$, in that there exist $I_n : (\alpha\ T^n)\ set$, $Abs_{S,n} : \alpha\ T^n \to \alpha\ S^n$ and $Rep_{T,n} : \alpha\ T^n \to \alpha\ S^n$ such that $type\_definition\ Abs_{S,n}\ Rep_{T,n}\ I_n$ holds.

A formal proof in Isabelle/HOL of the core of this theorem can be found in [11]; due to the HOL type system limitations, the formal proof is restricted to the case when $n = 2$, but also indicates how to iterate the argument for arbitrary n and shows the iterations for 3 and 4.

[4]Incidentally, in Isabelle/HOL the operator L is called *lists*—a suggestive name given that this operator is the set-based counterpart of the *list* type constructor: it takes any set A to the set of lists formed with elements of A.

Proof sketch. We proceed by induction on n. For $n = 1$, we simply take $I_n = I$, $Abs_{S,n} = Abs_S$ and $Rep_{S,n} = Rep_S$, so the desired fact holds by our assumptions.

For the induction step, assume that we have $I_n : (\alpha\ T^n)\ set$, $Abs_{S,n} : \alpha\ T^n \to \alpha\ S^n$ and $Rep_{T,n} : \alpha\ T^n \to \alpha\ S^n$ such that $type\_definition\ Abs_{S,n}\ Rep_{T,n}\ I_n$ holds. We define $I_{n+1} : (\alpha\ T^{n+1})\ set$, $Abs_{S,n+1} : \alpha\ T^{n+1} \to \alpha\ S^{n+1}$ and $Rep_{T,n+1} : \alpha\ T^{n+1} \to \alpha\ S^{n+1}$ as follows: $I_{n+1} \equiv Iset\ I_n$; $Abs_{S,n+1} = Abs_S \circ funOf\,(Trel\,(relOf\,Abs_{S,n}))$; $Rep_{S,n+1} = Rep_S \circ funOf\,(Trel\,(relOf\,Rep_{S,n}))$.

Above, the operators $relOf : (\alpha \to \beta) \to (\alpha \to \beta \to bool)$ and $funOf : (\alpha \to \beta \to bool) \to (\alpha \to \beta)$ provide back and forth conversions between bijective relations and (partially) bijective functins. Namely, we have the following properties for them, where $bij\_betw\ A\ B\ f$ says that the restriction of the function $f : \alpha \to \beta$ to $A : \alpha\ set$ is a bijection between A and $B : \beta\ set$: (1) If $bij\_betw\ A\ B\ f$, then $bijBetw\ A\ B\ (relOf\ f)$ and $funOf\,(relOf\ f) = f$; (2) If $bijBetw\ A\ B\ R$, then $bij\_betw\ A\ B\ (funOf\ R)$ and $relOf\,(funOf\ f R = R$.

To prove $type\_definition\ Abs_{S,n+1}\ Rep_{T,n+1}\ I_{n+1}$ compositionally (along the definitions of the abstraction ad represenations operators), we introduce a generalization of *type_definition* that does not assume one of its argument functions (namely the representation function) to operate on the entire domain, but on an additional parameter set. Namely, we define $bij\_pair : (\beta \to \alpha) \to (\alpha \to \beta) \to \beta\ set \to \alpha\ set \to bool$ as follows, where we highlight the differences from *type_definition*: $bij\_pair\ r\ a\ BA \equiv (\forall y.\ y \in B \longrightarrow r\ y \in A) \wedge (\forall x.\ x \in A \longrightarrow a\ x \in B) \wedge (\forall y : \beta.\ y \in B \longrightarrow a\,(r\ y) = y) \wedge (\forall x : \alpha.\ x \in A \longrightarrow r\,(a\ x) = x)$. Thus, $bij\_pair\ r\ a\ B\ A$ says that a and r are mutually inverse bijections between A and B; in particular, we have (3) $bij\_pair\ r\ a\ B\ A = type\_definition\ r\ a\ UNIV\ A$.

Next, we define the relational counterpart of *bij_pair*, namely $bijPair : (\beta \to \alpha \to bool) \to (\alpha \to \beta \to bool) \to \beta\ set \to \alpha\ set \to bool$, such that $bijPair\ P\ Q\ B\ A$ says that P and Q are mutually inverse (relational) bijections between A and B. The operators *bij_pair* and *bijPair* correspond to each other via the translations between functions and relations: (4) If $bij\_pair\ r\ a\ B\ A$ then $bijPair\ (relOf\ r)\ (relOf\ a)\ B\ A$; (5) If $bijPair\ P\ Q\ B\ A$ then $bij\_pair\ (funOf\ P)\ (funOf\ Q)\ B\ A$. They also commute with relation and function composition: (6) If $bij\_pair\ r\ a\ B\ A$ and $bij\_pair\ r'\ a'\ A\ C$ then $bij\_pair\ (r' \circ r)\ (a' \circ a)\ B\ C$; (7) If $bijPair\ P\ Q\ B\ A$ and $bij\_pair\ P'\ Q'\ A\ C$ then $bijPair\ (P \circ P')\ (Q \circ Q')\ B\ C$.

Finally, using (A1), (A3) and (A4), we can prove the crucial property that *Trel* "lifts" the *bijPair* property relative to *Iset*: (8) If $bijPair\ P\ Q\ A\ B$ then $bijPair\ (Trel\ A\ B\ P)\ (Trel\ B\ A\ Q)\ (Iset\ A)\ (Iset\ B)$. □

Note that the theorem asserts the existence of n-level predicates I_n and abstraction-representation pairs (Abs_n, Rep_n) which extend the original ones, I and (Abs, Rep). But the question arises on whether these are the "right" extensions. The answer relies only on the suitability of I_n, since once that is decided than *any* projection pair (Abs_n, Rep_n) satisfying $type\_definition\ Abs_{S,n}\ Rep_{T,n}\ I_n$ would do—mirroring the fact that in a type definition (at level 1) only I matters and any (Abs, Rep) satisfying $type\_definition\ Abs_S\ Rep_T\ I$ is as good as any other.

Now, concerning the suitability of I_n, we note from the proof that $I_n = Iset^n\ UNIV$; so it all hinges upon whether *Iset* is the "right" way of lifting sets of elements of α to sets of elements of $\alpha\ T$ that respects the intended meaning of the subset I of $\alpha\ T$. In other words, whatever concept I is supposed to represent, we want that *Iset* provides a correct relativization of that concept from the entire type α to a subset $A : \alpha\ set$. Our assumptions (A2)–(A4) are sanity properties for such a relativization, but whether this is the correct relativization needs to be established in each particular case—in other words, it is the responsability of the user of our framework to provide a correct and meaningful *Iset* operator. This is easily seen to be the case in our two examples, where the move from I to *Iset* clearly represents the move from distributions on the whole type to distributions on a given subset, and from distinct lists on the whole type to distinct lists on a given subset, respectively. Moreover, the scheme that worked for distinct lists (of intersecting with the support operator) works for essentially the same reason for any container type.

Finally, a remark on the generality of the theorem: While we have formulated it in reference to the (possibly iterated) nesting of a *single* type constructor, the construction and proof can be straightforwardly (albeit tediously) extended to cope with combining / nesting different type constructors (which can also have more than one argument).

5. Related Work and Future Work

Isomorphic transfer is an important topic in formal reasoning, and is one of the main motivations of major recent developments such as Homotopy Type Theory [16]. Transfer along ismorphisms, as well as along quotient projections and refinements, is also well represented in the world of HOL-based provers (and especially Isabelle/HOL), e.g., [17, 18, 19, 8, 20, 21]. In dependent type theory, problems similar to the ones we address here are formulated in the context of (partial) setoids [22, 23], and proof assistants such as Coq [24] (via SSReflect [25]) and Lean [26] offer quasi-automated mechanisms to address them, thus mitigating what is referred to as "setoid hell", i.e., the need to prove over and over again that certain (partial) equivalence relations are preserved by the defined functions. Note that the problem we addressed in this paper, which has to do with relativization to sets, is a particular case of relativization to partial equivalence relations—and in fact dealing with the former at higher-order types quickly escalates to having to deal with the latter, even in the absence of dependent types [10].

Another technical part of our setting concerns the HOL defined types, which are not included in their base types but only injected into them. Other formalisms offer pure subtyping / inclusion mechanisms, notably PVS [27], F* [28], and logical frameworks featuring refinement types [29]—where the goal of isomorphic transfer is replaced by corresponding goals concerning automating aspects of type checking.

Next, we will focus on the most closely related work, namely Isabelle's Lifting and Transfer package [30, 8]. From the very beginning, the authors of this tool have been aware of the potential difficulties raised by nested type constructors, and have implemented a solution based on parameterized transfer relations. For example, in the case of *distrib*, the current implementation defines automatically a relation in $(\alpha \to real) \to \beta\ distrib \to bool$ (thus using different type variables α and β) as the composition between the relator $\Rightarrow_{real}$ and the relational version of $Abs_{distrib}$. This does not guarantee an isomorphic relationship between nested applications of $(\_ \to real)$ and *distrib* like our framework does (exported as a *type_definition* theorem), but offers enough infrastructure in order to allow the user to "lift" the definition of constants from $(\_ \to real)$ to *distrib* even in the presence of nesting. Since the relativized defining set is not provided explicitly (like in our framework, as the operator *Iset*), the proof goals arising when transferring theorems containing such constants are quite intricate and convoluted, in sharp contrast to the non-nested case. On the other hand, at the expense of asking the user to provide more information (not just the relator *Trel* but also the operator *Iset*) and proving some sanity properties, our approach really flattens everything to the "first-order" non-nesting case—with the potential of simplifying the proof goals. Moreover, currently the transfer package relies on a parametricity theorem for the defined type's underlying predicate, which our approach does not. Overall, our approach requires some more initial effort from the user upon introducing a new type, but this has the potential to pay off in the later stages of the developments.

This having been said, our work addresses only a subproblem of the overall lifting and transfer problem, which the Lifting and Transfer package addresses quite comprehensively. So we do not envision our development as an alternative to this mature tool, but as a possible "add-on" that can improve the usability and automation of the tool's handling of nested types. Since what we produce for nested types are *type_definition* theorems, these can in principle be directly integrated into the Lifting and Transfer package (via the "setup_lifting" command), save for one difficulty caused by the fact that any provided abstraction operators are currently required to be registered as dataype-like constructors—addressing this formal engineering problem is ongoing work. (Of course, the integration will involve the fully general case, of different type constructors of possible multiple arguments nested in arbitrary ways.)

Another short-term plan is to provide some generic infrastructure that automates our inferred result for arbitrary type constructors, which can then be instantiated to different cases. This is being tackled (in joint work with Dmitriy Traytel) with the help of a quasi-foundational developement of polymorphic locales, generalizing Isabelle's standard (monomorphic) locales [31, 32] in a manner that does not impair the meta-properties of the logic and definitional mechanisms underlying Isabelle/HOL [33, 34, 35, 36].

Acknowledgments. We thank the three anonymous reviewers for their valuable comments and suggestions, which led to the improvement of the presentation and to the discussion of more related work.

References

[1] The HOL Team, The HOL4 Theorem Prover, 2024. Http://hol.sourceforge.net/.

[2] J. Harrison, HOL light: An overview, in: S. Berghofer, T. Nipkow, C. Urban, M. Wenzel (Eds.), TPHOLs 2009, volume 5674 of *LNCS*, Springer, 2009, pp. 60–66. doi:10.1007/978-3-642-03359-9_4.

[3] T. Nipkow, L. C. Paulson, M. Wenzel, Isabelle/HOL — A Proof Assistant for Higher-Order Logic, volume 2283 of *LNCS*, Springer, 2002. doi:10.1007/3-540-45949-9.

[4] S. Berghofer, M. Wenzel, Inductive datatypes in HOL – lessons learned in formal-logic engineering, in: Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin-Mohring, L. Théry (Eds.), TPHOLs 1999, volume 1690 of *LNCS*, Springer, 1999, pp. 19–36. doi:10.1007/3-540-48256-3_3.

[5] D. Traytel, A. Popescu, J. C. Blanchette, Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving, in: LICS 2012, IEEE Computer Society, 2012, pp. 596–605. doi:10.1109/LICS.2012.75.

[6] J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, D. Traytel, Truly modular (co)datatypes for isabelle/hol, in: G. Klein, R. Gamboa (Eds.), ITP 2014, volume 8558 of *LNCS*, Springer, 2014, pp. 93–110. doi:10.1007/978-3-319-08970-6_7.

[7] J. C. Blanchette, F. Meier, A. Popescu, D. Traytel, Foundational nonuniform (co)datatypes for higher-order logic, in: 32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017, IEEE Computer Society, 2017, pp. 1–12. URL: https://doi.org/10.1109/LICS.2017.8005071. doi:10.1109/LICS.2017.8005071.

[8] B. Huffman, O. Kuncar, Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL, in: G. Gonthier, M. Norrish (Eds.), CPP 2013, volume 8307 of *LNCS*, Springer, 2013, pp. 131–146. doi:10.1007/978-3-319-03545-1_9.

[9] O. Kunčar, A. Popescu, From types to sets by local type definition in higher-order logic, J. Autom. Reason. 62 (2019) 237–260. doi:10.1007/s10817-018-9464-6.

[10] A. Popescu, D. Traytel, Admissible types-to-pers relativization in Higher-Order Logic, Proc. ACM Program. Lang. 7 (2023) 1214–1245. URL: https://doi.org/10.1145/3571235. doi:10.1145/3571235.

[11] G. Buday, A. Popescu, Supplementary material associated to this paper, https://www.andreipopescu.uk/suppl/iFM2024.zip, 2024.

[12] R. Godement, Topologie algébrique et théorie des faisceaux, Publications de l'Institut de Mathématique de l'Université de Strasbourg, Hermann, Paris, 1958.

[13] E. Moggi, Notions of computation and monads, Inf. Comput. 93 (1991) 55–92. URL: https://doi.org/10.1016/0890-5401(91)90052-4. doi:10.1016/0890-5401(91)90052-4.

[14] J. C. Reynolds, Types, abstraction and parametric polymorphism, in: R. E. A. Mason (Ed.), IFIP 1983, North-Holland/IFIP, 1983, pp. 513–523.

[15] P. Wadler, Theorems for free!, in: J. E. Stoy (Ed.), FPCA 1989, ACM, 1989, pp. 347–359. doi:10.1145/99370.99404.

[16] T. Univalent Foundations Program, Homotopy Type Theory: Univalent Foundations of Mathematics, https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[17] P. V. Homeier, A design structure for higher order quotients, in: J. Hurd, T. F. Melham (Eds.), Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, volume 3603 of *Lecture Notes in Computer Science*, Springer, 2005, pp. 130–146.

[18] C. Kaliszyk, C. Urban, Quotients revisited for Isabelle/HOL, in: W. C. Chu, W. E. Wong, M. J. Palakal, C. Hung (Eds.), Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), TaiChung, Taiwan, March 21 - 24, 2011, ACM, 2011, pp. 1639–1644. URL: https://doi.org/10.1145/1982185.1982529. doi:10.1145/1982185.1982529.

[19] P. Lammich, Automatic data refinement, in: S. Blazy, C. Paulin-Mohring, D. Pichardie (Eds.), Interactive Theorem Proving - 4th International Conference, ITP, volume 7998 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 84–99.

[20] A. Schropp, A. Popescu, Nonfree datatypes in Isabelle/HOL - animating a many-sorted metatheory, in: G. Gonthier, M. Norrish (Eds.), Certified Programs and Proofs - Third International Conference,

CPP, volume 8307 of *LNCS*, Springer, 2013, pp. 114–130.

[21] M. Milehins, An extension of the framework types-to-sets for Isabelle/HOL, in: A. Popescu, S. Zdancewic (Eds.), CPP 2022, ACM, 2022, pp. 180–196. doi:10.1145/3497775.3503674.

[22] G. Barthe, V. Capretta, O. Pons, Setoids in type theory, J. Funct. Program. 13 (2003) 261–293. doi:10.1017/S0956796802004501.

[23] T. Altenkirch, S. Boulier, A. Kaposi, N. Tabareau, Setoid type theory - A syntactic translation, in: G. Hutton (Ed.), Mathematics of Program Construction - 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings, volume 11825 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 155–196. URL: https://doi.org/10.1007/978-3-030-33636-3_7. doi:10.1007/978-3-030-33636-3_7.

[24] Y. Bertot, P. Castéran, Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions, Texts in Theoretical Computer Science. An EATCS Series, Springer, 2004. URL: https://doi.org/10.1007/978-3-662-07964-5. doi:10.1007/978-3-662-07964-5.

[25] G. Gonthier, A. Mahboubi, An introduction to small scale reflection in coq, J. Formaliz. Reason. 3 (2010) 95–152. URL: https://doi.org/10.6092/issn.1972-5787/1979. doi:10.6092/ISSN.1972-5787/1979.

[26] L. M. de Moura, S. Kong, J. Avigad, F. van Doorn, J. von Raumer, The lean theorem prover (system description), in: A. P. Felty, A. Middeldorp (Eds.), CADE-25, volume 9195 of *LNCS*, Springer, 2015, pp. 378–388. doi:10.1007/978-3-319-21401-6_26.

[27] J. M. Rushby, S. Owre, N. Shankar, Subtypes for specifications: Predicate subtyping in PVS, IEEE Trans. Software Eng. 24 (1998) 709–720. URL: https://doi.org/10.1109/32.713327. doi:10.1109/32.713327.

[28] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Strub, M. Kohlweiss, J. K. Zinzindohoue, S. Z. Béguelin, Dependent types and multi-monadic effects in F, in: R. Bodík, R. Majumdar (Eds.), Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016, ACM, 2016, pp. 256–270. URL: https://doi.org/10.1145/2837614.2837655. doi:10.1145/2837614.2837655.

[29] W. Lovas, F. Pfenning, Refinement types for logical frameworks and their interpretation as proof irrelevance, Log. Methods Comput. Sci. 6 (2010). URL: https://doi.org/10.2168/LMCS-6(4:5)2010. doi:10.2168/LMCS-6(4:5)2010.

[30] O. Kunčar, 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] F. Kammüller, M. Wenzel, L. C. Paulson, Locales - A sectioning concept for Isabelle, in: Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin-Mohring, L. Théry (Eds.), Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings, volume 1690 of *Lecture Notes in Computer Science*, Springer, 1999, pp. 149–166. URL: https://doi.org/10.1007/3-540-48256-3_11. doi:10.1007/3-540-48256-3_11.

[32] C. Ballarin, Locales and locale expressions in isabelle/isar, in: S. Berardi, M. Coppo, F. Damiani (Eds.), Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers, volume 3085 of *Lecture Notes in Computer Science*, Springer, 2003, pp. 34–50. URL: https://doi.org/10.1007/978-3-540-24849-1_3. doi:10.1007/978-3-540-24849-1_3.

[33] O. Kuncar, A. Popescu, Comprehending Isabelle/HOL's consistency, in: H. Yang (Ed.), Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, volume 10201 of *Lecture Notes in Computer Science*, Springer, 2017, pp. 724–749. URL: https://doi.org/10.1007/978-3-662-54434-1_27. doi:10.1007/978-3-662-54434-1_27.

[34] O. Kuncar, A. Popescu, A consistent foundation for Isabelle/HOL, J. Autom. Reason. 62 (2019)

531–555. URL: https://doi.org/10.1007/s10817-018-9454-8. doi:10.1007/S10817-018-9454-8.

[35] O. Kuncar, A. Popescu, Safety and conservativity of definitions in HOL and Isabelle/HOL, Proc. ACM Program. Lang. 2 (2018) 24:1–24:26. URL: https://doi.org/10.1145/3158112. doi:10.1145/3158112.

[36] A. Gengelbach, T. Weber, Proof-theoretic conservative extension of HOL with ad-hoc overloading, in: V. K. I. Pun, V. Stolz, A. Simão (Eds.), Theoretical Aspects of Computing - ICTAC 2020 - 17th International Colloquium, Macau, China, November 30 - December 4, 2020, Proceedings, volume 12545 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 23–42. URL: https://doi.org/10.1007/978-3-030-64276-1_2. doi:10.1007/978-3-030-64276-1_2.