Nonfree Datatypes in Isabelle/HOL
Animating a Many-Sorted Metatheory

Andreas Schropp and Andrei Popescu

Technische Universität München, Germany

Abstract. Datatypes freely generated by their constructors are well supported in mainstream proof assistants. Algebraic specification languages offer more expressive datatypes on axiomatic means: nonfree datatypes generated from constructors modulo equations. We have implemented an Isabelle/HOL package for nonfree datatypes, without compromising foundations. The use of the package, and its nonfree iterator in particular, is illustrated with examples: bags, polynomials and λ-terms modulo α-equivalence. The many-sorted metatheory of nonfree datatypes is formalized as an ordinary Isabelle theory and is animated by the package into user-specified instances. HOL lacks a type of types, so we employ an ad hoc construction of a universe embedding the relevant parameter types.

1 Introduction

Free datatypes are at the heart of logic and computer science and are well supported in most proof assistants. Equational theories over them are often less convenient. Finite multisets or "bags" are a popular construction and can be regarded as finite lists modulo the permutation of elements. This results in the following nonfree datatype of bags over the type α, with "empty bag" and "bag-insert" constructors:

$$\begin{array}{l}\mathtt{datatype}\ \alpha\ \mathsf{bag} = \mathsf{BEmp} \mid \mathsf{BIns}\ \alpha\ (\alpha\ \mathsf{bag}) \\ \mathtt{where}\ \mathsf{BIns}\, a_1\ (\mathsf{BIns}\, a_2\ B) = \mathsf{BIns}\, a_2\ (\mathsf{BIns}\, a_1\ B)\end{array}$$

where the equation—left-commutativity—is (implicitly) universally quantified over a_1, a_2 and B. Bags are thus specified by list-like constructors and an identification of differently constructed terms based on (all consequences of) the indicated equation.

This style of definition is standard in the world of algebraic specifications [6, 7]. Nonfree datatypes and suitable recursors for them allow one to express many concepts at the appropriate level of abstraction, as opposed to encoding them in more concrete free types. For instance, bags are encodable as lists, but the price is a loss of abstraction, hence more error-prone processing methods. This is equally true for programming [25] and theorem proving. However, mainstream proof assistants based on type theory [1, 5] or higher-order logic (HOL) [10, 18] currently do not provide mechanisms for specifying nonfree datatypes directly.

In HOL-based provers, such as our favorite one Isabelle/HOL [18], datatypes are not integrated into the logic, but are provided as a definitional layer on top of the logical primitives. Given a user specification, a *definitional package* produces the appropriate types, terms and theorems, including induction and recursion schemes. In this paper, we present a definitional package in Isabelle/HOL for nonfree datatypes. Its expressiveness goes a little beyond standard algebraic specifications (typically, equational theories), allowing Horn clauses over equations *and predicates*.

Our package also contributes a new methodology for addressing an old problem: the incomplete, dynamic nature of typical package certification. Indeed, the mathematics behind a datatype package requires reasoning about arbitrary numbers of types and operators on them. This is not possible generically inside HOL, because it lacks a type of types. The constructions performed by HOL packages are usually certified dynamically, for each particular instance that the user requests. Our package essentially limits the amount of dynamic certification to a minimum of uniform facts concerning the transfer across isomorphisms.[1] The nontrivial part of the constructions is statically certified in a metatheory formalized in Isabelle. It is parameterized on a collection of sets over a fixed "universe" type, instead of a collection of types. This "universe" type is instantiated by ad hoc sums over the relevant types when animating the metatheory.

The paper is structured in two main parts. The first part, consisting of §2, illustrates the package by examples—bags, polynomials and λ-terms modulo α-equivalence—carefully chosen to illustrate different aspects and features of the package: nonfree recursion, interaction with Isabelle's type classes, and predicate-based Horn specifications. We also hope that these examples help popularize nonfree recursion, a standard technique that is not so standard in proof assistants. The second part describes the package design and architecture: §3 illustrates on an example the actual steps that are automated by the package, §4 presents the formalization of the metatheory up to the construction of the initial model, and §5 shows how the metatheory is automatically instantiated to user-specified datatypes. The appendix gives more examples and details. The package is compatible with Isabelle2013 and is publicly available [23].

Preliminaries. In this paper, by HOL we mean classical higher-order logic with Hilbert choice, schematic polymorphism and the typedef principle. The Isabelle/HOL proof assistant [18] is an implementation of HOL enhanced with Haskell-style type classes [9] and locales [15]. Types in HOL are either atomic types such as unit, nat and bool, or type variables α, β, or built from these using type constructors. We use postfix notation for type constructors, e.g., $\alpha\ \mathsf{list}$ and $\alpha\ \mathsf{set}$ denote the list and powerset types over α. Polymorphic types are not syntactically distinguished—e.g., $\alpha\ \mathsf{list}$ also denotes the polymorphic type $\forall\alpha.\ \alpha\ \mathsf{list}$. We write $\alpha \to \beta$, $\alpha + \beta$, and $\alpha \times \beta$ for the function-space, sum and product types, respectively. All types are nonempty.[2] New types are introduced with the typedef principle by carving out nonempty subsets of existing types. A term t of type τ is indicated as $t : \tau$. (We prefer the more mathematical notations $\alpha \to \beta$ and $t : \tau$ to the Isabelle notations $\alpha \Rightarrow \beta$ and $t :: \tau$.)

Type classes are an overloading mechanism wired into Isabelle's type system. A type class C specifies for its member types, $\tau : C$, constants of composite types containing τ and axioms for these constants. Typical cases are the algebraic classes, e.g., $\tau : \mathsf{semigroup}$ means that there exists an operation $+ : \tau \to \tau \to \tau$ assumed associative. Isabelle locales are essentially proof contexts, fixing type and term variables with assumptions. A locale can be instantiated by providing concrete types and terms for its type and term variables and then discharging its assumptions. This makes the instantiated content of the locale available in the outer context.

[1] Additionally we employ rewriting steps, forward chaining of facts, well-sortedness checking rules, and finite datatypes and functions over them to construct the signature instantiation.

[2] HOL is not following the propositions-as-types paradigm, so this is not troublesome.

2 The Package in Action

Here we present the package and its different features by examples. We start with the datatype of bags, whose single-equation specification makes it easy to present in detail the package's contract: what is expected from the user and what is produced in response.

2.1 Bags

The declaration of the datatype of bags from §1 produces the type α bag and the following polymorphic constants:
– the constructors $\mathsf{BEmp} : \alpha\ \mathsf{bag}$ and $\mathsf{BIns} : \alpha \to \alpha\ \mathsf{bag} \to \alpha\ \mathsf{bag}$,
– the iterator $\mathsf{iter\_bag} : \beta \to (\alpha \to \beta \to \beta) \to \alpha\ \mathsf{bag} \to \beta$.

In addition, several characteristic theorems are derived. They include facts also available for standard free datatypes:
– Case distinction: $(B = \mathsf{BEmp} \longrightarrow \varphi) \wedge (\forall a\ C.\ B = \mathsf{BIns}\ a\ C \longrightarrow \varphi) \longrightarrow \varphi$
– Induction: $\varphi\ \mathsf{BEmp} \wedge (\forall a\ B.\ \varphi\ B \to \varphi\ (\mathsf{BIns}\ a\ B)) \longrightarrow (\forall B.\ \varphi\ B)$

Note that the injectivity of the constructors, here,

$$\mathsf{BIns}\ a_1\ B_1 = \mathsf{BIns}\ a_2\ B_2 \longrightarrow a_1 = a_2 \wedge B_1 = B_2,$$

is not among these facts, since it does not hold for nonfree datatypes.

The interesting derived theorems are those specific to nonfree datatypes:
– The characteristic equation(s) specified by the user:
$\mathsf{BIns}\ a_1\ (\mathsf{BIns}\ a_2\ B) = \mathsf{BIns}\ a_2\ (\mathsf{BIns}\ a_1\ B)$
– Conditional equations for iteration:
$\mathsf{bag\_alg}\ E\ I \to \mathsf{iter\_bag}\ E\ I\ \mathsf{BEmp} = E$
$\mathsf{bag\_alg}\ E\ I \to (\forall a\ B.\ \mathsf{iter\_bag}\ E\ I\ (\mathsf{BIns}\ a\ B) = I\ a\ (\mathsf{iter\_bag}\ E\ I\ B))$
where $\mathsf{bag\_alg}\ E\ I$ is the predicate $\forall\ a_1\ a_2\ b.\ I\ a_1\ (I\ a_2\ b) = I\ a_2\ (I\ a_1\ b)$.

Thus, the package produces a type α bag that satisfies the specified equation. In addition, α bag is *initial* among the algebras $(\beta, E : \beta, I : \alpha \to \beta \to \beta)$ satisfying the equation (with E and I replacing BEmp and BIns) as expressed by the predicate $\mathsf{bag\_alg}\ E\ I$. This means that from α bag to any such algebra there exists precisely one morphism, i.e., function commuting with the algebra operations. The existence of a morphism is expressed by the iteration equations: given such an algebra, the morphism is $\mathsf{iter\_bag}\ E\ I$. Its uniqueness is given by the induction principle.

As with other definitional packages for recursion, the user does not needs to employ the iterator directly—the package allows the user to inline I and E in the desired recursive equations. For example, the following specifies the map function for bags:

$$\begin{aligned}\texttt{nonfreerec}\ \mathsf{bag\_map} : (\alpha \to \beta) \to \alpha\ \mathsf{bag} \to \beta\ \mathsf{bag}\ \texttt{where}\\ \mathsf{bag\_map}\ f\ \mathsf{BEmp} &= \mathsf{BEmp}\\ \mathsf{bag\_map}\ f\ (\mathsf{BIns}\ a\ B) &= \mathsf{BIns}\ (f\ a)\ (\mathsf{bag\_map}\ f\ B)\end{aligned}$$

In response to this command, the package does the following (for a fixed $f : \alpha \to \beta$):
(1) identifies E and I as being $\mathsf{BEmp} : \beta\ \mathsf{bag}$ and $(\lambda a.\ \mathsf{BIns}\ (f\ a)) : \alpha \to \beta\ \mathsf{bag} \to \beta\ \mathsf{bag}$;
(2) defines $\mathsf{bag\_map}\ f = \mathsf{iter\_bag}\ E\ I$;
(3) prompts the user to discharge the goal $\mathsf{bag\_alg}\ E\ I$;
(4) infers the desired *unconditional* equations stated in the `nonfreerec` declaration from the conditional equations for iter_bag and the fact proved at step (3).

Thus, the user obtains the desired simplification rules for the newly introduced bag_map after discharging the bag_alg goal, here,

$$\mathsf{BIns}\ (f\ a_1)\ (\mathsf{BIns}\ (f\ a_2)\ B)\ =\ \mathsf{BIns}\ (f\ a_2)\ (\mathsf{BIns}\ (f\ a_1)\ B)$$

which is immediate from the characteristic equation for β bag.

This complication, of having to discharge goals that imply well-definedness of a function definition, is inherent in the nature of quotiented types and is shared with the quotient and nominal packages [11, 13, 14]. For this paper's examples, the conditions are easy to discharge by simplification (but this cannot be guaranteed in general). This is also the case for the sum of a numeric function over the elements of a bag:

$$\begin{aligned}\texttt{nonfreerec}\ \mathsf{sum} : (\alpha \to \mathsf{nat}) \to \alpha\ \mathsf{bag} \to \mathsf{nat}\ \texttt{where}\\ \mathsf{sum}\ f\ \mathsf{BEmp}\ &=\ 0\\ \mathsf{sum}\ f\ (\mathsf{BIns}\ a\ B)\ &=\ \mathsf{sum}\ f\ B + f\ a\end{aligned}$$

which yields the goal $(m + f\ a_1) + f\ a_2 = (m + f\ a_2) + f\ a_1$. It is discharged using associativity and commutativity of $+$ on nat, which means that the definition generalizes: we can replace nat with the type class member β : comm_monoid_add, covering all types equipped with a commutative monoid structure $(\beta, 0, +)$. The multiplicity of an element in a bag, $\mathsf{mult} : \alpha \to\ \alpha\ \mathsf{bag} \to \mathsf{nat}$ is obtained as $\mathsf{mult}\ a\ B = \mathsf{sum}\ (\lambda a'.\ \text{if } a = a' \text{ then } 1 \text{ else } 0)\ B$.

2.2 Algebra

The package can be used to streamline algebraic constructions. The following example builds the ring of polynomials over a commutative ring α with variables in β, where Sc is the embedding of scalars yielding Sc 0 as the zero polynomial and Var gives the polynomial variables. (§B.2 shows another standard construction, sum of algebras.)

$$\begin{aligned}&\texttt{datatype}\ (\alpha : \mathsf{comm\_ring}, \beta)\ \mathsf{poly}\ =\ \mathsf{Sc}\ \alpha\ |\ \mathsf{Var}\ \beta\ |\ \mathsf{Uminus}\ ((\alpha,\beta)\ \mathsf{poly})\ |\\ &\quad \mathsf{Plus}\ ((\alpha,\beta)\ \mathsf{poly})\ ((\alpha,\beta)\ \mathsf{poly})\ |\ \mathsf{Times}\ ((\alpha,\beta)\ \mathsf{poly})\ ((\alpha,\beta)\ \mathsf{poly})\\ &\texttt{where}\ (-a) = b\ \longrightarrow\ \mathsf{Uminus}\ (\mathsf{Sc}\ a) = \mathsf{Sc}\ b\\ &\quad \texttt{and}\ a_1 + a_2 = a\ \longrightarrow\ \mathsf{Plus}\ (\mathsf{Sc}\ a_1)\ (\mathsf{Sc}\ a_2) = \mathsf{Sc}\ a\\ &\quad \texttt{and}\ a_1 * a_2 = a\ \longrightarrow\ \mathsf{Times}\ (\mathsf{Sc}\ a_1)\ (\mathsf{Sc}\ a_2) = \mathsf{Sc}\ a\\ &\quad \texttt{and}\ \mathsf{Plus}\ (\mathsf{Sc}\ 0)\ P\ =\ P\\ &\quad \texttt{and}\ \mathsf{Plus}\ (\mathsf{Plus}\ P_1\ P_2)\ P_3\ =\ \mathsf{Plus}\ P_1\ (\mathsf{Plus}\ P_2\ P_3)\\ &\text{*** Etc.: All the commutative-ring axioms for Plus, Times, Sc 0 ***}\end{aligned}$$

This example illustrates the nontrivial use of type class annotations in the datatype declaration: since α is a ring, it provides operations $*$, $+$, 0, which are used in the definition of the new type. Type class constraints in polymorphic datatype specifications are already present in Isabelle's standard datatype package, but only serve as a syntactic constraint there. The feature is essential here for performing universal extensions over an unspecified algebraic structure: we need to form a type depending on its operations.

The first three clauses ensure that the restrictions of polynomial inverse, addition and multiplication to scalars, collapse to the scalar operations $-$, $+$ and $*$. They illustrate the use of parameters from type α. Strictly speaking each of the clauses forms an infinite family of Horn clauses, indexed by either $a : \alpha$ or $a_1, a_2 : \alpha$. One may employ any condition on the parameters, not just equality as here.

This direct definition of polynomials can replace the tedious standard construction based on lists. By its characteristic equations, (α,β) poly forms a commutative ring if α does and we can register this with the type-class system. Universality is established by an operator that extends morphisms $f : (\alpha : \mathsf{comm\_ring}) \to (\gamma : \mathsf{comm\_ring})$ (assumed to commute with $+, *, 0$) and variable interpretations $g : \beta \to \gamma$, to morphisms $\mathsf{ext}\ f\ g : (\alpha,\beta)\ \mathsf{poly} \to \gamma$. In the context of such f and g, we define ext by simply writing down its desired interaction with the polynomial operators:

$$\begin{array}{l}\texttt{nonfreerec}\ \mathsf{ext} : (\alpha,\beta)\ \mathsf{poly} \to \gamma\ \texttt{where}\\ \mathsf{ext}\ (\mathsf{Sc}\ a) = f\ a \qquad \mathsf{ext}\ (\mathsf{Var}\ b) = g\ b \qquad \mathsf{ext}\ (\mathsf{Uminus}\ P) = -\ \mathsf{ext}\ P\\ \mathsf{ext}\ (\mathsf{Plus}\ P\ Q) = \mathsf{ext}\ P + \mathsf{ext}\ Q \qquad \mathsf{ext}\ (\mathsf{Times}\ P\ Q) = \mathsf{ext}\ P * \mathsf{ext}\ Q\end{array}$$

where simplification with the ring axioms of γ and the morphism axioms of f immediately discharges the goals. Polynomial evaluation is obtained from ext taking $f = \mathsf{id}$.

2.3 λ-terms modulo α-equivalence

Next we discuss a less standard example—λ-terms modulo α-equivalence—which employs the full expressive power of the package, combining parameter conditions with Horn predicates. This type can be specified as the initial model of a Horn theory if we factor in the freshness predicate and at least one of the substitution and swapping operators [19, 21]. In particular, the following provides (a type isomorphic to) the λ-calculus terms (modulo α-equivalence) over variables in α and constants in β, including the syntactic constructors, freshness and substitution:

$$\begin{array}{l}\texttt{datatype}\ (\alpha,\beta)\ \mathsf{lterm} = \mathsf{Var}\ \alpha \mid \mathsf{Ct}\ \beta \mid \mathsf{App}\ ((\alpha,\beta)\ \mathsf{lterm})\ ((\alpha,\beta)\ \mathsf{lterm}) \mid\\ \qquad\qquad \mathsf{Lam}\ \alpha\ ((\alpha,\beta)\ \mathsf{lterm}) \mid \mathsf{Subst}\ ((\alpha,\beta)\ \mathsf{lterm})\ ((\alpha,\beta)\ \mathsf{lterm})\ \alpha\\ \texttt{with}\ \mathsf{fresh} : \alpha \to (\alpha,\beta)\ \mathsf{lterm} \to \mathsf{bool}\\ \texttt{where}\ \ (\mathsf{Var}\ x)\ [t/x] = t\\ \quad \texttt{and}\ \ x \neq y \longrightarrow (\mathsf{Var}\ y)\ [t/x] = \mathsf{Var}\ y\\ \quad \texttt{and}\ \ (\mathsf{Ct}\ c)\ [t/x] = \mathsf{Ct}\ c\\ \quad \texttt{and}\ \ (\mathsf{App}\ s_1\ s_2)\ [t/x] = \mathsf{App}\ (s_1[t/x])\ (s_2[t/x])\\ \quad \texttt{and}\ \ x \neq y \wedge \mathsf{fresh}\ y\ t \longrightarrow (\mathsf{Lam}\ y\ s)\ [t/x] = \mathsf{Lam}\ y\ (s\ [t/x])\\ \quad \texttt{and}\ \ x \neq y \longrightarrow \mathsf{fresh}\ x\ (\mathsf{Var}\ y)\\ \quad \texttt{and}\ \ \mathsf{fresh}\ x\ (\mathsf{Ct}\ c)\\ \quad \texttt{and}\ \ \mathsf{fresh}\ x\ s_1 \wedge \mathsf{fresh}\ x\ s_2 \longrightarrow \mathsf{fresh}\ x\ (\mathsf{App}\ s_1\ s_2)\\ \quad \texttt{and}\ \ \mathsf{fresh}\ x\ (\mathsf{Lam}\ x\ s)\\ \quad \texttt{and}\ \ \mathsf{fresh}\ x\ s \longrightarrow \mathsf{fresh}\ x\ (\mathsf{Lam}\ y\ s)\\ \quad \texttt{and}\ \ x \neq y \wedge \mathsf{fresh}\ x\ s \longrightarrow \mathsf{Lam}\ y\ s = \mathsf{Lam}\ x\ (s\ [\mathsf{Var}\ x\ /\ y])\end{array}$$

where we wrote $s[t/x]$ instead of $\mathsf{Subst}\ s\ t\ x$. Besides operations, this type also comes with a predicate fresh, which plays a crucial role in the behavior of the capture-free substitution operators, as regulated by the above Horn clauses. Specifically, substitution can "enter" λ-abstractions only under certain freshness conditions. Nevertheless, substitution can always be reduced away from terms by using the last clause to perform a renaming to a fresh variable.

This Horn-based definition of λ-terms is easily extendable to any syntax with static bindings, but does require some tuning to become a useful framework for reasoning about bindings. In particular it lacks a substitution-free induction schema. One type

of task where the Horn view of λ-terms excels are recursive definitions: besides going through modulo α-equivalence, they also yield compositionality with freshness and substitution as a bonus. This is argued in [21] with many examples, ranging from higher-order abstract syntax and semantic-domain interpretation to CPS transformations. These examples are instances of the nonfree recursion provided by our package.

For instance, $\mathsf{occs}\ t\ x$ yields the number of free occurrences of a variable x in a λ-term t. It is defined stating the "naive" recursive equations (as if terms were not factored to α) together with indicating the correct behavior w.r.t. freshness and substitution:

$$\mathtt{nonfreerec}\ \mathsf{occs} : (\alpha,\beta)\ \mathsf{lterm} \to (\alpha \to \mathsf{nat})\ \mathtt{where}$$
$$\mathsf{occs}\ (\mathsf{Ct}\ c) = (\lambda x.\ 0) \qquad \mathsf{occs}\ (\mathsf{Lam}\ y\ s) = (\lambda x.\ \text{if } x = y \text{ then } 0 \text{ else } \mathsf{occs}\ s\ x)$$
$$\mathsf{occs}\ (\mathsf{Var}\ y) = (\lambda x.\ \text{if } x = y \text{ then } 1 \text{ else } 0) \qquad \mathsf{occs}\ (\mathsf{App}\ s\ t) = (\lambda x.\ \mathsf{occs}\ s\ x + \mathsf{occs}\ s\ y)$$
$$\mathsf{occs}\ (s\ [t/y]) = (\lambda x.\ \mathsf{occs}\ s\ y * \mathsf{occs}\ t\ x + (\text{if } x = y \text{ then } 0 \text{ else } \mathsf{occs}\ s\ x))$$
$$\mathsf{fresh}\ y\ s \longrightarrow \mathsf{occs}\ s\ y = 0$$

Note that, while the operators require (recursive) *equations*, predicates such as fresh require *implications*. Indeed, the implication for fresh indicates that, on the target domain $\alpha \to \mathsf{nat}$, freshness is interpreted as $\lambda y\ s.\ \mathsf{occs}\ s\ y = 0$. The goals emerging from this definition amount to arithmetic properties known by the Isabelle simplifier.

3 Automated Constructions

Here we sketch the development required to obtain the functionality provided by the package, using our λ-term example. (1) One starts with the free datatype of "pre-terms":

$$\mathtt{datatype}\ (\alpha,\beta)\ \mathsf{lterm}' = \mathsf{Var}'\ \alpha \mid \mathsf{Ct}'\ \beta \mid \mathsf{App}'\ ((\alpha,\beta)\ \mathsf{lterm}')\ ((\alpha,\beta)\ \mathsf{lterm}') \mid \mathsf{Lam}'\ \alpha\ ((\alpha,\beta)\ \mathsf{lterm}') \mid \mathsf{Subst}'\ ((\alpha,\beta)\ \mathsf{lterm}')\ ((\alpha,\beta)\ \mathsf{lterm}')\ \alpha$$

(2) Next, one defines mutually inductively the desired "equality" $\equiv$ and the "pre-fresh" predicate. (In general, mutually recursive datatypes involve n equalities, one for each type, and m predicates, one for each predicate specified by the user.)

$\mathtt{inductive}\ \equiv : \alpha\ \mathsf{lterm}' \to \alpha\ \mathsf{lterm}' \to \mathsf{bool}\ \mathtt{and}\ \mathsf{fresh}' : \alpha\ \mathsf{lterm}' \to \mathsf{bool}$
$\mathtt{where}$
*** One clause for each user-specified Horn clause: ***
$\quad (\mathsf{Var}\ x)[t/x] \equiv t$
$\quad \mathtt{and}\quad x \neq y \longrightarrow (\mathsf{Var}\ y)\ [t/x] \equiv \mathsf{Var}'\ y$
*** etc. ***
*** The equivalence rules: ***
$\quad \mathtt{and}\quad s \equiv s \quad \mathtt{and}\quad s_1 \equiv s_2 \longrightarrow s_2 \equiv s_1 \quad \mathtt{and}\quad s_1 \equiv s_2 \wedge s_2 \equiv s_3 \longrightarrow s_1 \equiv s_3$
*** A congruence rule for each user-specified constructor: ***
$\quad \mathtt{and}\quad s_1 \equiv t_1 \wedge s_2 \equiv t_2 \longrightarrow \mathsf{App}'\ s_1\ s_2 \equiv \mathsf{App}'\ t_1\ t_2$
*** etc. ***
*** A preservation rule for each constructor-predicate combination: ***
$\quad \mathtt{and}\quad s_1 \equiv t_1 \wedge s_2 \equiv t_2 \wedge \mathsf{fresh}'\ x\ (\mathsf{App}'\ s_1\ s_2) \longrightarrow \mathsf{fresh}'\ x\ (\mathsf{App}'\ t_1\ t_2)$
*** etc. ***

(3) The type α lterm is defined by quotienting α lterm$'$ by the equivalence $\equiv$, establishing a surjection $\pi : \alpha\ \mathsf{lterm}' \to \alpha\ \mathsf{lterm}$, with the choice function $\varepsilon : \alpha\ \mathsf{lterm} \to$

α lterm$'$ as its right inverse. The operations Var, App and Lam and the predicate fresh are defined on α lterm from the corresponding ones from α lterm$'$ using π and ε. (4) The induction principle from α lterm$'$ is transported to α lterm. (5) α lterm is shown to satisfy all the desired Horn clauses. To obtain the recursion principle, one fixes a type β with operations and relations on it and assumes it satisfies the Horn clauses. (6) A function $f : \alpha$ lterm$' \to \beta$ is then defined by standard recursion. (7) By induction on the derivation of $\equiv$, we get that f is invariant under equivalent arguments. (8) This allows one to define a function $g : \alpha$ lterm $\to \beta$ such that $g \circ \pi = f$. (9) Using the surjectivity of π, this function is shown to commute with the operations and preserve the relations.

All the involved constructions and proofs are fairly easy to perform by hand, but quite tedious and time-consuming. Parts (3–5) and (8,9) of this process can be eased by existing Isabelle quotient/lifting/transfer packages [12, 14].

Our package automates the whole construction. Moreover, it does not perform this construction over and over, for each newly specified nonfree datatype. We have experimented with a different methodology:
– Formalize the metatheory for an arbitrary many-sorted signature and Horn theory.
– Upon a user specification, instantiate the metathoery, then copy isomorphically the relevant types, operations, and theorems about them.

The next two sections describe these steps.

4 Formalized Metatheory

We have formalized the theory of Horn clauses up to the construction of the initial model. The development is parameterized by an arbitrary signature (giving sorts and sorted operations and relation symbols) and an arbitrary Horn theory over the signature, i.e., a set of Horn clauses. Both terms and clauses are deeply embedded. Sorts represent relevant Isabelle types. A specific feature of our formalization is the consideration of parameters and parameter conditions in clauses, motivated by the desire to capture parameterized instances such as polymorphic datatypes and clausal side conditions.

We will use the following constants. Inl : $\alpha \to \alpha + \beta$ and Inr : $\beta \to \alpha + \beta$ are the left and right injections into the sum type, and isInl, isInr : $\alpha + \beta \to$ bool are their corresponding selectors; namely, isInl c holds iff c has the form Inl a for some a, and isInr c holds iff c has the form Inr b for some b. [] is the empty list, $[a_1, \ldots, a_n]$ is the list of the n indicated elements. map : $(\alpha \to \beta) \to \alpha$ list $\to \beta$ list is the standard list-map operator, and map2 : $(\alpha \to \beta \to \gamma) \to \alpha$ list $\to \beta$ list $\to \gamma$ list is its binary counterpart, with map2 $f\ [a_1, \ldots, a_n]\ [b_1, \ldots, b_n] = [f\ a_1\ b_1, \ldots, f\ a_n\ b_n]$. Similarly, list_all : $(\alpha \to$ bool$) \to \alpha$ list $\to$ bool is the universal quantifier over lists, with list_all $\varphi\ [a_1, \ldots, a_n]$ meaning that $\varphi\ a_i$ holds for all i, and list_all2 : $(\alpha \to \beta \to$ bool$) \to \alpha$ list $\to \beta$ list $\to$ bool is its binary counterpart, with list_all2 φ *al bl* meaning that *al* has the form $[a_1, \ldots, a_n]$, *bl* has the from $[b_1, \ldots, b_n]$, and $\varphi\ a_i\ b_i$ holds for all i. In particular list_all2 φ *al bl* requires that *al* and *bl* have equal lengths. As a notational convention, we use the suffix "l" to indicate lists. E.g., if *ps* ranges over the type psort, then *psl* ranges over psort list.

4.1 Horn Clause Syntax

We define the types var, of variables, and pvar, of parameter variables (p-variables), as copies of nat. Our constructions are parameterized by the following type variables:

sort, of sorts, giving the syntactic categories of terms (representing the mutually recursive datatypes); opsym, of operation symbols (representing the datatype constructors); rlsym, of relation symbols (representing relations); param, the parameter universe ; psort, of parameter sorts (p-sorts; representing the parameter types in the datatype).

The type of terms is defined as follows:

```
datatype (sort, opsym) trm = Var sort var |
                             Op opsym (pvar list) (((sort, opsym) trm) list)
```

Thus a term T is either a sorted variable Var s x or has the form Op σ *pxl Tl*, applying an operation symbol σ to a list *pxl* of parameter variables and a list *Tl* of terms.

The type of atoms (or atomic statements) is defined as follows:

```
datatype (sort, opsym, rlsym, psort, param) atm =
    Pcond (param list → bool) (psort list) (pvar list) |
    Eq ((sort, opsym) trm) ((sort, opsym) trm) |
    Rl rlsym (pvar list) (((sort, opsym) trm) list)
```

We provide an intuition of the semantics of these atoms here. §4.3 provides the details. The semantics of these atoms is relative to interpretations of sorts as subsets of a model, of variables as elements in a model, of operation symbols as functions on a model, of relation symbols as relations on a model and of p-variables as parameters:
(1) Parameter-condition atoms have the form Pcond *R psl pxl*. Semantically they will be interpreted as the predicate *R* on the interpretation of the p-variables *pxl* (where this interpretation is assumed to be consistent with the p-sorts *psl*).
(2) Equational atoms have the form Eq s T_1 T_2. They will be interpreted as a specialized "equality" relation between T_1 and T_2, assumed to be of sort s.
(3) Relational atoms have the form Rl π *pxl Tl*. They will be interpreted as the model relation corresponding to π on the interpretations of the p-variables *pxl* and the interpretations of the terms *Tl*. The sorts of *Tl* are assumed to agree with the sorting of π.

Horn clauses are essentially lists of atoms: the premises are paired with one atom, the conclusion. In §4.3 we will interpret a Horn clause as the implication between the interpretations of the premises and the conclusion, schematically quantified over variable interpretations:

```
datatype (sort, opsym, rlsym, psort, param) hcl =
    Horn (((sort, opsym, rlsym, psort, param) atm) list)
         ((sort, opsym, rlsym, psort, param) atm)
```

In what follows, we fix the type parameters and omit them when writing the various types that depend on them, e.g., writing trm instead of (sort, opsym) trm.

4.2 Signatures

We define signatures as a locale that fixes the data required to classify terms and parameters according to sorts:

```
locale Signature =
    fixes stOf : opsym → sort
    and   arOf : opsym → sort list    and   arOfP : opsym → psort list
    and   rarOf : rlsym → sort list   and   rarOfP : rlsym → psort list
    and   params : psort → param → bool
    and   prels : ((param list → bool) × psort list) set
```

Recall from the definition of terms that operation symbols are applied not only to terms, but also to parameters. Then arOf (read "arity of"), arOfP (read "parameter-arity of") and stOf (read "sort of"), regulate the sorts of terms (or, in general, elements of models) and parameters that an operation symbol takes and the sort of terms it returns. Similarly, rarOf and rarOfP indicate the arities and parameter-arities of relation symbols. Moreover, params classifies parameters according to sorts. Finally, prels specifies the set of relations over parameters that can be used as parameter conditions in Horn clauses, together with their intended arities. Given $(R, psl) \in \mathsf{prels}$, we only care about the behavior of R on lists pl of parameters having sorts psl according to params, i.e., such that list_all2 params pl psl holds. We have to represent R as a relation on the larger type param list because dependent types are not available. Similar phenomena are observable in our definitions of models below.

4.3 Models

We work in the Signature context. The (well-formed) terms of a given sort are defined as the predicate $\mathsf{trms} : \mathsf{sort} \to \mathsf{trm} \to \mathsf{bool}$, by requiring that operation symbols are applied according to their arities.

A model is a tuple $(\alpha, intSt, intOp, intRl)$, where:
– α is the carrier type,
– $intSt : \mathsf{sort} \to \alpha \to \mathsf{bool}$ classifies the elements of α according to sorts;
– $intOp : \mathsf{opsym} \to \mathsf{param\ list} \to \alpha\ \mathsf{list} \to \alpha$ interprets the operation symbols as parameterized operations on α;
– $intRl : \mathsf{rlsym} \to \mathsf{param\ list} \to \alpha\ \mathsf{list} \to \mathsf{bool}$ interprets the relation symbols as parameterized relations on α.

In (well-formed) models the interpretation of operation symbols has to be compatible with sorting, i.e., the following predicate compat $intSt$ $intOp$ holds:

$$\forall \sigma\ pl\ al.\ \mathsf{list\_all2\ params}\ (\mathsf{arOfP}\ \sigma)\ pl \wedge \mathsf{list\_all2}\ intSt\ (\mathsf{arOf}\ \sigma)\ al \to intSt\ (\mathsf{stOf}\ \sigma)\ (intOp\ \sigma\ pl\ al).$$

Given a model $(\alpha, intSt, intOp, intRl)$, the notions of term interpretation and atom satisfaction are defined relative to interpretations of parameter variables $intPvar : \mathsf{psort} \to \mathsf{pvar} \to \mathsf{param}$ and variables $intVar : \mathsf{sort} \to \mathsf{var} \to \alpha$. For equational atoms, we do not require equality, but further parameterize on a relation $intEq : \alpha \to \alpha \to \mathsf{bool}$.

$$\mathsf{intTrm}\ intOp\ intPvar\ intVar\ (\mathsf{Var}\ s\ x) = intVar\ s\ x$$
$$\mathsf{intTrm}\ intOp\ intPvar\ intVar\ (\mathsf{Op}\ \sigma\ pxl\ Tl) = intOp\ \sigma\ (\mathsf{map2}\ intPvar\ (\mathsf{arOfP}\ \sigma)\ pxl)\ (\mathsf{map}\ (\mathsf{intTrm}\ intOp\ intPvar\ intVar)\ Tl)$$

$$\mathsf{satAtm}\ intOp\ intEq\ intRl\ intPvar\ intVar\ (\mathsf{Pcond}\ R\ psl\ pxl) \leftrightarrow R\ (\mathsf{map2}\ intPvar\ psl\ pxl)$$
$$\mathsf{satAtm}\ intOp\ intEq\ intRl\ intPvar\ intVar\ (\mathsf{Eq}\ s\ T_1\ T_2) \leftrightarrow intEq\ (\mathsf{intTrm}\ intOp\ intPvar\ intVar\ T_1)\ (\mathsf{intTrm}\ intOp\ intPvar\ intVar\ T_2)$$
$$\mathsf{satAtm}\ intOp\ intEq\ intRl\ intPvar\ intVar\ (\mathsf{Rl}\ \pi\ pxl\ Tl) \leftrightarrow intRl\ \pi\ (\mathsf{map2}\ intPvar\ (\mathsf{rarOfP}\ \pi)\ pxl)\ (\mathsf{map}\ (\mathsf{intTrm}\ intOp\ intPvar\ intVar)\ Tl)$$

Thus, the term interpretation is defined recursively over terms, employing interpretations of p-variables and variables. For atom satisfaction, we distinguish the three kinds of atom, employing the parameter-conditions, the equality interpretation and the relation-symbol interpretation, respectively. Note that the interpretations do not depend on the model-carrier sorting $intSt : \alpha \to \mathsf{sort}$. However, for well-formed models we

prove that well-sorted interpretations of (p-)variables yield term interpretations compatible with sorting, in that they send terms of sort s to model elements of sort s:

lemma: compat *intSt intOp* $\wedge$ $(\forall$ *ps px*. params *ps* (*intPvar ps px*)) $\wedge$
$(\forall$ *s x*. *intSt s* (*intVar s x*)) $\rightarrow$ (trms *s T* $\rightarrow$ *intSt s* (intTrm *intOp intPvar intVar T*)).

The above approach is pervasive in our formalization: We do not index everything by sorts, but use global (unsorted) functions and relations as much as possible, and then show that they are compatible with sorting. This optimization is particularly helpful when we factor terms to the Horn-induced equivalence relation building a single quotient instead of a sorted family of quotients (as customary in universal algebra).

Finally, satisfaction of a Horn clause by a model is defined as the implication between satisfaction of the premises and satisfaction of the conclusion for all well-sorted interpretations *intPvar* of the p-variables and *intVar* of the variables:

satHcl *intSt intOp intEq intRl* (Horn *atml atm*) $\leftrightarrow$
$\forall$ *intPvar intVar*. $(\forall$ *ps px*. params *ps* (*intPvar ps px*)) $\wedge$ $(\forall$*s x*. *intSt s* (*intVar s x*)) $\wedge$
list_all (satAtm *intOp intEq intRl intPvar intVar*) *atml* $\rightarrow$
satAtm *intOp intEq intRl intPvar intVar atm*

4.4 The Initial Model of a Horn Theory

Traditionally, ground terms are simply terms with no free variables. However, in our parameterized setting, terms contain p-variables, while the ground terms will need to contain actual parameters. We define a separate type of ground terms, gtrm, built recursively from operation symbols applied to lists of parameters and list of ground terms:
datatype (opsym, param) gtrm = Gop opsym (param list) (((opsym, param) gtrm)list)

The initial model of a Horn theory will be constructed by factoring ground terms to an equivalence relation. Hence its carrier will be the following type of "Horn terms" defined to be sets of ground terms:

type_synonym (opsym, param) htrm = ((opsym, param) gtrm) set

In what follows, we fix a signature with assumptions guaranteeing non-emptiness of sorts and p-sorts and a well-formed Horn theory HCL. Technically, we work in the context of the following locale extending the Signature locale:
locale HornTheory = Signature + fixes HCL : hcl set
assumes $\forall$hcl $\in$ HCL. wf hcl and $\forall s$. reach s and $\forall ps.\ \exists p.$ params *ps p*

Above, wf hcl states that the Horn clause is well-formed in that all its atoms are well-formed in the expected way, e.g., in equational atoms Eq $s\ T_1\ T_2$, s is the sort of T_1 and T_2. The inductively defined predicate reach s states that the sort s is reachable by operation symbols. This ensures the existence of ground terms of sort s, where sorting of ground terms gtrms : sort $\rightarrow$ gtrm $\rightarrow$ bool is defined as expected.

On gtrm we define mutually inductive relations Geq : gtrm $\rightarrow$ gtrm $\rightarrow$ bool and Grel : rlsym $\rightarrow$ param list $\rightarrow$ gtrm list $\rightarrow$ bool in a similar fashion to the example of §3, but working symbolically with the clauses in HCL instead of concrete clauses. We show that Geq is an equivalence and that both relations are compatible with sorting and with the operations. This allows us to quotient gtrm by Geq, giving the type htrm. We lift the sorting gtrms of ground terms and the interpretations Gop, Grel of the operation and relation symbols on ground terms to equivalence classes. This yields the functions

htrms : sort → htrm → bool, Hop : opsym → param list → htrm list → htrm and Hrel : rlsym → param list → htrm list → bool.

The ground-term model (gtrm, gtrms, Gop, Grel) satisfies all the clauses in HCL if we interpret equality as Geq:

lemma: hcl ∈ HCL → satHcl gtrms Gop Geq Grel hcl

From this, we obtain that the Horn-term model (htrm, htrms, Hop, Hrel) satisfies the clauses with the standard interpretation of equality:

theorem satisfaction: hcl ∈ HCL → satHcl htrms Hop (=) Hrel hcl

Structural induction is easily inherited by Horn terms from ground terms:

theorem induction: (∀σ *pl Hl*. list_all2 params (arOfP σ) *pl* ∧
list_all2 htrms (arOf σ) *Hl* ∧ list_all2 φ (arOf σ) *Hl* → φ (stOf σ) (Hop σ *pl Hl*))
→ (htrms *s H* → φ *s H*).

Moreover, the cases theorem is obtained as a degenerate induction. We are left to show that (htrm, htrms, Hop, Hrel) is initial among the models of HCL. First we define giter : (opsym → param list → α list → α) → gtrm → α that interprets ground terms with an operation symbol interpretation on a type α, as giter *intOp* (Gop σ *pl Tl*) = *intOp* σ *pl* (map (giter *intOp*) *Tl*). Then we lift giter to htrm equivalence classes, giving iter : (opsym → param list → α list → α) → htrm → α. If (α, *intSt*, *intOp*, *intRl*) is a model that satisfies HCL, then iter *intOp* is well-sorted and behaves like an iterator, i.e., commutes with the operations, and preserves the relations:

theorem it_sort: compat *intSt intOp* ∧ (∀ hcl ∈ HCL. satHcl *intSt intOp* (=) *intRl* hcl)
→ htrms *s H* → *intSt s* (iter *intOp H*)

theorem iteration: compat *intSt intOp* ∧ (∀ hcl ∈ HCL. satHcl *intSt intOp* (=) *intRl* hcl)
→ iter *intOp* (Hop σ *pl Hl*) = *intOp* σ *pl* (map (iter *intOp*) *Hl*)

theorem it_pres: compat *intSt intOp* ∧ (∀ hcl ∈ HCL. satHcl *intSt intOp* (=) *intRl* hcl)
→ Hrel π *pl Hl* → *intRl* π *pl* (map (iter *intOp*) *Hl*).

After some lemmas concerning the interaction between the choice function and the operations on Gop, the above theorems are proved by induction on the definition of Geq and Grel. Note that our iterator only depends on the operation part of the model, although its properties rely on the whole model and its satisfaction of HCL.

5 Animation of the Metatheory

From a purely mathematical viewpoint, having formalized the general case for arbitrary signatures and Horn theories, we did capture all the instances. But we have to bridge the gap between the abstract characterization of the instances in the metatheory and the instance descriptions offered by users of the package. Moreover, the metatheory introduces operations over a quotient term universe, while users want to use curried datatype constructors between distinguished types for each of the mutually recursive datatypes.

5.1 Instantiation of the Metatheory

We focus on an example instantiation of the metatheory here and refer to the appendix for an overview of the instantiation in general.

To obtain the λ-terms modulo α from §2.3, we simply instantiate the `HornTheory` locale. The types are instantiated as follows:
– sort becomes a type with 1 element, lt, for the unique syntactic category of λ-terms;
– opsym becomes a type with 5 elements, var, ct, app, lam, subst, corresponding to the operations Var, Ct, App, Lam, Subst;
– rlsym becomes a type with 1 element, fr, corresponding to the predicate fresh;
– param becomes the sum type $\alpha + \beta$, embedding the type α of variables and β of constants used in λ-terms and thus forming the parameter universe;
– psort becomes a type with 2 elements, a and b, matching the 2 kinds of parameters.

The signature variables are instantiated as follows:
– stOf _ = lt; arOf var = []; arOfP var = [a]; arOf ct = []; arOfP ct = [b];
arOf app = [lt, lt]; arOfP app = []; arOf lam = [lt]; arOfP lam = [a];
arOf subst = [lt, lt]; arOfP subst = [a]; rarOf fr = [lt]; rarOfP fr = [a];
– params $ps\ p \longleftrightarrow (ps = \mathsf{a} \wedge \mathsf{isInl}\ p) \vee (ps = \mathsf{b} \wedge \mathsf{isInr}\ p)$;
– prels = $\{(\mathsf{dif}_2, [\mathsf{a}, \mathsf{a}])\}$, where dif_2 is the function sending any list of two parameters of the form $[\mathsf{Inl}\ a_1, \mathsf{Inl}\ a_2]$ to (the truth-value of) $a_1 \neq a_2$ (and with immaterial definition elsewhere).

Finally, HCL is instantiated to the set containing the reflections of the λ-term clauses. For example, $x \neq y \wedge \mathsf{fresh}\ x\ s \longrightarrow \mathsf{Lam}\ y\ s = \mathsf{Lam}\ x\ (s\ [\mathsf{Var}\ x\ /\ y])$ becomes $\mathsf{Horn}\ [\mathsf{atm}_1, \mathsf{atm}_2]\ \mathsf{atm}_3$, where:
– we take x and y to be distinct elements of pvar and s to be some element of var;
– $\mathsf{atm}_1 = \mathsf{Pcond}\ \mathsf{dif}_2\ [\mathsf{a}, \mathsf{a}]\ [x, y]$,
– $\mathsf{atm}_2 = \mathsf{Rl}\ \mathsf{fr}\ [x]\ [\mathsf{Var}\ \mathsf{lt}\ s]$ and $\mathsf{atm}_3 = \mathsf{Eq}\ \mathsf{lt}\ T_1\ T_2$, with $T_1 = \mathsf{Op}\ \mathsf{lam}\ [y]\ [\mathsf{Var}\ \mathsf{lt}\ s]$ and $T_2 = \mathsf{Op}\ \mathsf{lam}\ [x]\ [\mathsf{Op}\ \mathsf{subst}\ [y]\ [\mathsf{Var}\ \mathsf{lt}\ s,\ \mathsf{Op}\ \mathsf{var}\ [x]\ []]]$.

Then, after checking the `HornTheory` assumptions for this particular instances, we indeed obtain valid formulations of the satisfaction, induction and iteration theorems for λ-terms as instances of the general theorems. However, these formulations are inconvenient to use in a theorem prover. One would certainly prefer to write $\mathsf{App}\ s_1\ s_2$ instead of $\mathsf{Hop}\ \mathsf{app}\ []\ [s_1, s_2]$ for λ-term application, and $\forall x\ y\ s.\ x \neq y \wedge \mathsf{fresh}\ x\ s \longrightarrow \mathsf{Lam}\ y\ s = \mathsf{Lam}\ x\ (s\ [\mathsf{Var}\ x\ /\ y])$ instead of $\mathsf{satHcl}\ \mathit{intSt}\ \mathit{intOp}\ (=)\ \mathit{intRl}\ (\mathsf{Horn}\ [\mathsf{atm}_1, \mathsf{atm}_2]\ \mathsf{atm}_3)$.

Superficially, fixing this seems to be a matter of syntactic sugar. But the situation is a little more complex, since we also want to use a more appropriate type for λ-terms. Indeed, htrm may contain junk—the general theorems only speak about sorted terms. Therefore, the type we care about needs to be carved out from htrm by restricting to those T such that $\mathsf{htrms}\ \mathsf{lt}\ T$ (where lt is here the only sort). Then App needs to be defined as a copy of Hop app on the new type, also using two arguments instead of lists with two elements. These transformations are realized with the isomorphic transfer of types and terms, which we describe in the next section.

5.2 Isomorphic Transfer

Isomorphic transfer is based on establishing appropriate bijections between primitive types, lifting these bijections to composite types and mapping term constructions under the corresponding bijections away from the input types.

We shall employ relators, which are operators on predicates matching the type constructors. E.g., given $\varphi : A \to \mathsf{bool}$ and $\psi : B \to \mathsf{bool}$, $\varphi \otimes \psi : A \times B \to \mathsf{bool}$ is defined

by $(\varphi \otimes \psi)\ (a, b) \leftrightarrow (\varphi\ a \wedge \psi\ b)$ and $\varphi \Rightarrow \psi : (A \to B) \to \mathsf{bool}$ is defined by $(\varphi \Rightarrow \psi)\ f \leftrightarrow (\forall a.\ \varphi\ a\ \to\ \psi\ (f\ a))$.

$\mathsf{htrms\ lt} : \mathsf{htrm} \to \mathsf{bool}$	lterm
$\mathsf{isInl} : \mathsf{param} \to \mathsf{bool}$	α
$\mathsf{isInr} : \mathsf{param} \to \mathsf{bool}$	β
$(\mathsf{list\_all2\ params\ (arOfP\ app)}) \otimes (\mathsf{list\_all2\ htrms\ (arOf\ app)}) :$ $\mathsf{param\ list} \times \mathsf{htrm\ list} \to \mathsf{bool}$	$\mathsf{lterm} \times \mathsf{lterm}$
$(\mathsf{list\_all2\ params\ (arOfP\ lam)}) \otimes (\mathsf{list\_all2\ htrms\ (arOf\ lam)}) :$ $\mathsf{param\ list} \times \mathsf{htrm\ list} \to \mathsf{bool}$	$\alpha \times \mathsf{lterm}$

Fig. 1. Instance types and predicates (left) versus target types (right)

Figure 1 shows two categories of types side by side:
– on the left, the *instance types*, i.e., those obtained from the locale instantiation, where necessary together with predicates describing the relevant subset based on the sorting;
– on the right, the corresponding *target types* exported to the user.

We assume α and β have been fixed and omit spelling them out, e.g., we write lterm instead of $(\alpha, \beta)\mathsf{lterm}$. Also, param, htrm, etc. refer to the concrete types obtained by the locale instantiation from §5.1.

The first 3 rows show the primitive types. For the Horn terms, we have defined lterm by carving out from trmHCL the terms of sort lt (were there multiple sorts, we would have multiple target types of terms). For parameters, the instance type was defined from the target types, as their sum. In either case, we have bijections between sets of elements in the instance types satisfying corresponding predicates and the target types.

These bijections are extended to bijections between the domains of the instance operations and the intended domains of the target operations[3]—rows 4 and 5 show the extensions for two operation symbols, app and lam. To see how the extension operates, note that the instance predicates regulate the length of the lists and the sorts of their contents. E.g., since $\mathsf{arOf\ app} = [\mathsf{lt}, \mathsf{lt}]$, we see that $\mathsf{list\_all}\ (\mathsf{arOf\ app})\ Hl$ requires that Hl have the form $[H_1, H_2]$ such that $\mathsf{htrms\ lt}\ H_1$ and $\mathsf{htrms\ lt}\ H_2$ hold—thus, the lists boil down to pairs of Horn terms of sort lt, hence correspond bijectively to $\mathsf{lterm} \times \mathsf{lterm}$.

With the bijection construction in place, we proceed to copy the operations on the instance types into operations on the target types, by defining constants equal to their image under the corresponding bijection. E.g., $\mathsf{App} : \mathsf{lterm} \times \mathsf{lterm} \to \mathsf{lterm}$ is defined as the image of $\mathsf{Hop\ app} : \mathsf{param\ list} \times \mathsf{htrm\ list} \to \mathsf{trmHCL}$ restricted according to the suitable predicates. Thus, App corresponds to $\mathsf{Hop\ app}$ under the lifted bijection to $\mathsf{lterm} \times \mathsf{lterm} \to \mathsf{lterm}$ from the set of elements of $\mathsf{param\ list} \times \mathsf{htrm\ list} \to \mathsf{trmHCL}$ for which the predicate $(\mathsf{list\_all2\ params\ (arOfP\ app)}) \otimes (\mathsf{list\_all2\ htrms\ (arOf\ app)}) \Rightarrow \mathsf{htrms\ (stOf\ app)}$ holds. This set contains $\mathsf{Hop\ app}$ because of the sorting of app.

Finally, the theorems about instance types, that is, the satisfaction, induction, cases and recursion theorems, are transported from the instance types to the target types. Technically this works because we choose the bijection on propositions to be the identity. For instance, let us consider the induction theorem, where we write I_2 instead of

[3] To ease the presentation, we ignore currying and pretend that the domains are products.

list_all2:
$\forall\varphi\ s\ H.\ (\forall\sigma\ pl\ Hl.\ \mathsf{l}_2\ \mathsf{params}\ (\mathsf{arOfP}\ \sigma)\ pl\ \wedge\ \mathsf{l}_2\ \mathsf{htrms}\ (\mathsf{arOf}\ \sigma)\ Hl\ \wedge\ \mathsf{l}_2\ \varphi\ (\mathsf{arOf}\ \sigma)\ Hl \rightarrow \varphi\ (\mathsf{stOf}\ \sigma)\ (\mathsf{Hop}\ \sigma\ pl\ Hl)) \rightarrow \mathsf{htrms}\ s\ H \rightarrow \varphi\ s\ H$
To ease the presentation let us pretend that the signature only has app and lam as operation symbols. The theorem is processed as follows, into equivalent theorems. First, the quantification over σ is replaced by conjunction over all operation symbols:
$\forall\varphi\ s\ H.\ (\forall\varphi\ pl\ Hl.\ \mathsf{l}_2\ \mathsf{params}\ (\mathsf{arOfP}\ \mathsf{app})\ pl \wedge \mathsf{l}_2\ \mathsf{htrms}\ (\mathsf{arOf}\ \mathsf{app})\ Hl \wedge \mathsf{l}_2\ \varphi\ (\mathsf{arOf}\ \mathsf{app})\ Hl \rightarrow \varphi\ (\mathsf{stOf}\ \mathsf{app})\ (\mathsf{Hop}\ \mathsf{app}\ pl\ Hl))$
$\wedge\ (\forall\varphi\ pl\ Hl.\ \mathsf{l}_2\ \mathsf{params}\ (\mathsf{arOfP}\ \mathsf{lam})\ pl \wedge \mathsf{l}_2\ \mathsf{htrms}\ (\mathsf{arOf}\ \mathsf{lam})\ Hl \wedge \mathsf{l}_2\ \varphi\ (\mathsf{arOf}\ \mathsf{lam})\ Hl \rightarrow \varphi\ (\mathsf{stOf}\ \mathsf{lam})\ (\mathsf{Hop}\ \mathsf{lam}\ pl\ Hl))$
$\rightarrow \mathsf{htrms}\ s\ H \rightarrow \varphi\ s\ H$
Computing the values of the sort and arity functions, this becomes:
$\forall\varphi\ s\ H.\ (\forall\varphi\ pl\ Hl.\ \mathsf{l}_2\ \mathsf{params}\ []\ pl\ \wedge\ \mathsf{l}_2\ \mathsf{htrms}\ [\mathsf{lt},\mathsf{lt}]\ Hl\ \wedge\ \mathsf{l}_2\ \varphi\ [\mathsf{lt},\mathsf{lt}]\ Hl \rightarrow \varphi\ \mathsf{lt}\ (\mathsf{Hop}\ \mathsf{app}\ pl\ Hl))$
$\wedge\ (\forall\varphi\ pl\ Hl.\ \mathsf{l}_2\ \mathsf{params}\ [\mathsf{a}]\ pl\ \wedge\ \mathsf{l}_2\ \mathsf{htrms}\ [\mathsf{lt}]\ Hl\ \wedge\ \mathsf{l}_2\ \varphi\ [\mathsf{lt}]\ Hl \rightarrow \varphi\ \mathsf{lt}\ (\mathsf{Hop}\ \mathsf{lam}\ pl\ Hl))$
$\rightarrow \mathsf{htrms}\ s\ H \rightarrow \varphi\ s\ H$
By isomorphic transfer over the aforementioned extended bijections, we obtain:
$(\forall H_1\ H_2.\ \varphi\ H_1\ \wedge\ \varphi\ H_2 \rightarrow \varphi\ (\mathsf{App}\ H_1\ H_2))\ \wedge\ (\forall x\ H.\ \varphi\ H\ \rightarrow\ \varphi\ (\mathsf{Lam}\ x\ H))\ \rightarrow\ \varphi\ H.$

In this step the l_2 params, l_2 htrms constraints have disappeared, since the extended bijections map the constrained variables *pl*, *Hl* to empty tuples, pairs or single elements.

5.3 General Animation Infrastructure

All these constructions, namely, defining the types and terms necessary for the instantiation, establishing bijections between primitive types, extending them to the relevant composite types, and transferring the term constructions and theorems to the target types, are automated by employing a general infrastructure for algorithmic rule systems and forward propagation of facts.

Algorithmic rule systems are collections of proven rules about moded judgments, which are defined predicates in Isabelle/HOL. The definition of such a judgment constitutes its propositional meaning, while the rules are theorems that constitute the sound algorithm we use to synthesize the outputs and establish the judgment. The appendix attempts to give a taste of this and for details we refer to the first author's M.Sc. thesis [22]. We just note here that the animation of algorithmic rule systems can be regarded as a deterministic variant of Lambda-Prolog [16].

We employ the new concept of "forward rules" to drive the instantiation of the metatheory and invoke the term transformations. A forward rule is an implicational theorem that, algorithmically speaking, waits for input facts matching its conjunctive head premise, processes them with algorithmic rule systems indicated by judgmental premises, issues term and type definitions indicated by further premises and makes output facts available.

Isomorphic transfer is implemented in the form of an algorithmic rule system. The appendix contains a simplified version and [22] the details. We want to note that currying of functions over finite products is an ad hoc higher-order transformation overriding the uniform transfer of applications. In our case the products are realized as lists over a universe. Currying an operator application $f\ (\mathsf{Cons}\ t\ ts)$ proceeds by recursion on the list argument, regarding the uncurry-image of the partially-curried operator $\psi_1\ f$ applied to the transformed first component $\psi_2\ t$, as the new operator $\psi_3^{-1}\ ((\psi_1\ f)\ (\psi_2\ t))$ in the recursive transfer of $\psi_3^{-1}\ ((\psi_1\ f)\ (\psi_2\ t))\ ts$. The general approach using algorithmic rule systems is beneficial for term transformations with nonuniform behaviour.

6 Conclusions and Related Work

We implemented the first package for nonfree datatypes in a HOL-based prover, pioneering a metatheory approach. We provide parameter conditions, relations, induction, case distinction, satisfaction of the specification and iterative recursion (i.e. initiality). The presented ideas are relevant to all HOL-based provers, but type class constraints are Isabelle-specific and essential for some nonfree datatypes (see §2.2).

The metatheories of packages in HOL usually are of an informal nature and rely on the dynamic checking of inferences for soundness. Formalizing their metatheories will make theorem provers more reliable by offering completeness guarantees. Metatheorems of a common shape can be processed uniformly, which leads to better extensibility of packages. Metatheory-based constructions are a relatively recent idea even in dependent type theories that can engage in generic programming over type universes [3]. Application of these metatheories is usually not facilitated with automated isomorphic transfer and is thus left to idealistic users.

The Isabelle package for (co)datatypes [26] based on bounded natural functors (BNFs) lacks support for equational theories. But nonfree datatypes defined with our package can be registered as a BNF and nested in later (co)datatype definitions.

Nonfree datatypes are natively supported by algebraic-specification provers such as the Maude ITP [2]. One simply declares signatures and arbitrary sets of equations in Maude, on top of which a basic mechanism for inductive reasoning is available. New function symbols can be declared together with equations defining them, but there is no compatibility check w.r.t. the other equations. This means a check of well-definedness as for our nonfree recursor is lacking.

Moca [4] translates nonfree datatype specifications with an equational theory specified in a extension of OCaml, down to implementation datatypes with private datatype constructors. These can only be used for pattern matching and inhabitants are instead constructed with construction functions that normalize w.r.t. the equational theory. Efficient construction functions are a core concern of Moca. A translation to Coq is planned.

The quotient/lifting/transfer packages of Isabelle [12, 14] overlap in functionality with our tool for isomorphic transfer. The novelty here is its realization inside a general infrastructure and the possibility of ad hoc higher-order transformations such as currying of functions on finite products. We support the transfer under setoid isomorphisms, so quotient lifting is available with the canonical surjection into the quotient type as the setoid isomorphism. Packages for quotient lifting/transfer can ease some parts of the manual construction of nonfree datatypes, see section 3.

In the homotopy interpretation of type theory there is a recent trend [24] to investigate "higher inductive datatypes" that feature constructors introducing equalities. The main motivation here is to represent constructions of homotopy theory by describing their path space, but quotients similar to our package can also be introduced. The univalence axiom implies [8] that isomorphic mathematical structures are identified, so isomorphic transfer is available by substitution.

Acknowledgements. We thank Tobias Nipkow for making this collaboration possible, Jasmin Blanchette and Armin Heller for commenting on a draft, Ondrej Kuncar for answering questions about Isabelle's new lifting/transfer package and the people on the Coq-Club mailing list for pointing us to related work.

References

1. The Coq Proof Assistant, 2013. http://coq.inria.fr.
2. Maude ITP, 2013. http://maude.cs.uiuc.edu/tools/itp.
3. T. Altenkirch, C. McBride, and P. Morris. Generic programming with dependent types. In *SSDGP*, pp. 209–257, 2006.
4. F. Blanqui, T. Hardin, and P. Weis. On the implementation of construction functions for non-free concrete data types. In *ESOP*, pp. 95–109, 2007.
5. A. Bove and P. Dybjer. Dependent types at work. In *LerNet ALFA Summer School*, pp. 57–99, 2008.
6. M. Clavel, F. J. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. The Maude system. In *RTA*, pp. 240–243, 1999.
7. CoFI task group on semantics, CASL — The Common Algebraic Specification Language, Semantics. www.brics.dk/Projects/CoFI/Documents/CASL, 1999.
8. T. Coquand and N. A. Danielsson. Isomorphism is equality. Draft, 2013.
9. F. Haftmann and M. Wenzel. Constructive type classes in Isabelle. In *TYPES*, pp. 160–174, 2006.
10. J. Harrison. HOL Light: A tutorial introduction. In *FMCAD*, pp. 265–269, 1996.
11. P. V. Homeier. A design structure for higher order quotients. In *TPHOLs*, pp. 130–146, 2005.
12. B. Huffman and O. Kuncar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In *Isabelle Users Workshop*, 2012.
13. B. Huffman and C. Urban. Proof pearl: A new foundation for Nominal Isabelle. In *ITP '10*, pp. 35–50, 2010.
14. C. Kaliszyk and C. Urban. Quotients revisited for Isabelle/HOL. In *SAC*, pp. 1639–1644, 2011.
15. F. Kammüller, M. Wenzel, and L. C. Paulson. Locales—a sectioning concept for Isabelle. In *TPHOLs*, pp. 149–166, 1999.
16. G. Nadathur and D. Miller. An overview of Lambda-Prolog. In *ICLP/SLP*, pp. 810–827, 1988.
17. T. Nipkow and L. C. Paulson. Proof pearl: Defining functions over finite sets. In *TPHOLs*, pp. 385–396, 2005.
18. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002.
19. M. Norrish. Recursive function definition for types with binders. In *TPHOLs*, pp. 241–256, 2004.
20. F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *CADE*, pp. 202–206, 1999.
21. A. Popescu and E. L. Gunter. Recursion principles for syntax with bindings and substitution. In *ICFP*, pp. 346–358, 2011.
22. A. Schropp. Instantiating deeply embedded many-sorted theories into HOL types in Isabelle. Master's thesis, Technische Universität München, 2012. `http://home.in.tum.de/~schropp/master-thesis.pdf`.
23. A. Schropp and A. Popescu. Nonfree datatypes: metatheory, implementation and examples. `http://sourceforge.net/projects/nonfree-data/files/bundle.zip/download`.
24. M. Shulman, D. Licata, P. L. Lumsdaine, et al. Higher inductive types on the homotopy type theory blog. `http://homotopytypetheory.org/category/higher-inductive-types/`.
25. V. Tannen and R. Subrahmanyam. Logical and computational aspects of programming with sets/bags/lists. In *ICALP*, pp. 60–75, 1991.
26. D. Traytel, A. Popescu, and J. C. Blanchette. Foundational, compositional (co)datatypes for higher-order logic—Category theory applied to theorem proving. In *LICS 2012*, pp. 596–605. 2012.

Appendix

This appendix gives more details, including more examples of nonfree datatypes defined with our package and more details on our tool for isomorphic transfer. In case the paper is accepted, all references to the appendix from the main paper will be replaced with references to a technical report made available online.

A Rough edges of the package

The package currently has a couple of rough edges, which we plan to remove soon. We only support iteration on nonfree datatypes, not "generalized recursion", i.e. using both the recursive result and the recursive argument. Generalized recursion is reducible to iteration, but its direct support would be more convenient. Moreover, the parameter conditions in Horn clauses are currently expected to be constants. This is a mere convenience issue, since by defining auxiliary constants we can cover arbitrary terms.

B More Examples

B.1 Finite Sets

Finite sets are perhaps the most famous nonfree datatype. We revisit this well-studied type to show how our package can be used to quickly prototype new iterators corresponding to different algebraic views. Existing iterators are also easily definable by nonfree recursion. In fact, we found nonfree recursion appropriate for defining pretty much every basic operator we could think on and between finite sets and bags.

Finite sets can be specified as initial algebra in a variety of ways. Thus, the following defines finite sets by further constraining bags to left-idempotence: [4]

$$\begin{array}{l}\texttt{datatype}\ \alpha\ \mathsf{fset} = \mathsf{Emp} \mid \mathsf{Ins}\ \alpha\ (\alpha\ \mathsf{fset})\\ \texttt{where}\ \mathsf{Ins}_1\texttt{:}\ \mathsf{Ins}\, a\, (\mathsf{Ins}\, b\, A) = \mathsf{Ins}\, b\, (\mathsf{Ins}\, a\, A)\\ \quad \texttt{and}\ \mathsf{Ins}_2\texttt{:}\ \mathsf{Ins}\, a\, (\mathsf{Ins}\, a\, A) = \mathsf{Ins}\, a\, A\end{array}$$

This representation facilitates the definition of back and forth operators between finite sets and bags. Thus, since finite sets are more constrained than bags, the following flattening operator goes through immediately:

$$\begin{array}{l}\texttt{nonfreerec}\ \mathsf{flat} : \alpha\ \mathsf{bag} \to \alpha\ \mathsf{fset}\ \texttt{where}\\ \qquad \mathsf{flat}\ \mathsf{BEmp} = \mathsf{Emp}\\ \qquad \mathsf{flat}\ (\mathsf{BIns}\, b\, B) = \mathsf{Ins}\, b\, (\mathsf{flat}\, B)\end{array}$$

Backwards, the following embeds sets into bags by considering multiplicity 1 for each element:

$$\begin{array}{l}\texttt{nonfreerec}\ \mathsf{emb} : \alpha\ \mathsf{fset} \to \alpha\ \mathsf{bag}\ \texttt{where}\\ \qquad \mathsf{flat}\ \mathsf{Emp} = \mathsf{BEmp}\\ \qquad \mathsf{flat}\ (\mathsf{Ins}\, a\, A) = \text{if}\ \mathsf{mult}\, a\, (\,\mathsf{emb}\, A) = 0\ \text{then}\ \mathsf{BIns}\, a\, (\mathsf{emb}\, A)\ \text{else}\ \mathsf{emb}\, A\end{array}$$

[4] Ins_1 and Ins_2 are names optionally given to the constraints.

If we further consider membership in, and cardinality of, a finite set, $\mathsf{mem} : \alpha \to \alpha\ \mathsf{fset} \to \mathsf{bool}$ and $\mathsf{card} : \alpha\ \mathsf{fset} \to \mathsf{nat}$ (also definable by nonfree recursion), then, by bag- and set- induction and simplification, we can prove several basic facts on the operations on bags and sets and their connections, such as:

- $\mathsf{mem}\ a\ A \leftrightarrow \mathsf{Ins}\ a\ A = A$
- $\mathsf{mem}\ a\ (\flat\ A) \leftrightarrow \mathsf{mult}\ a\ A \neq 0$
- $\mathsf{mult}\ a\ (\mathsf{emb}\ A) \neq 0 \leftrightarrow \mathsf{mem}\ a\ A$

With a little care, performing recursive definitions and proofs on nonfree structure can become similar to working with totally free structures: nonfree recursion merely requires an extra invocation of the Isabelle simplifier.

Next we look at alternative description of finite sets. The alternative insert- and union- views we discuss below similarly apply to lists and bags [25]. The following defines α-finite-sets as the semilattice freely generated by α:

$$\begin{array}{l}\texttt{datatype}\ \alpha\ \mathsf{fset}' = \mathsf{Emp}' \mid \mathsf{Singl}\ \alpha \mid \mathsf{Un}\ (\alpha\ \mathsf{fset}')\ (\alpha\ \mathsf{fset}') \\ \texttt{where}\ \texttt{Asc:}\ \mathsf{Un}\ (\mathsf{Un}\ A_1\ A_2)\ A_3 = \mathsf{Un}\ A_1\ (\mathsf{Un}\ A_2\ A_3) \\ \quad \texttt{and}\ \texttt{Cmt:}\ \mathsf{Un}\ A_1\ A_2 = \mathsf{Un}\ A_2\ A_1 \\ \quad \texttt{and}\ \texttt{Idm:}\ \mathsf{Un}\ A\ A = A \\ \quad \texttt{and}\ \texttt{Idt:}\ \mathsf{Un}\ \mathsf{Emp}\ A = A\end{array}$$

If we are after nonempty finite sets only, we can remove one constructor and one equation from above, obtaining a purely ACI characterization:

$$\begin{array}{l}\texttt{datatype}\ \alpha\ \mathsf{nfset} = \mathsf{Singl}\ \alpha \mid \mathsf{Unn}\ (\alpha\ \mathsf{nfset})\ (\alpha\ \mathsf{nfset}) \\ \texttt{where}\ \texttt{Asc:}\ \mathsf{Unn}\ (\mathsf{Unn}\ A_1\ A_2)\ A_3 = \mathsf{Unn}\ A_1\ (\mathsf{Unn}\ A_2\ A_3) \\ \quad \texttt{and}\ \texttt{Cmt:}\ \mathsf{Unn}\ A_1\ A_2 = \mathsf{Unn}\ A_2\ A_1 \\ \quad \texttt{and}\ \texttt{Idm:}\ \mathsf{Unn}\ A\ A = A\end{array}$$

The above views yield iterators corresponding to a purely algebraic approach. On the other hand, fold functionals in proof assistants try to minimize the conditions to be verified, e.g., by removing the idempotence and unit properties. Isabelle's library iterator [17] should ideally be defined as follows, in the context of fixed type β and associative-commutative function $f : \beta \to \beta \to \beta$:

$$\begin{array}{l}\texttt{nonfreerec}\ \mathsf{ifold} : (\alpha \to \beta) \to \beta \to \alpha\ \mathsf{fset} \to \beta\ \texttt{where} \\ \qquad \mathsf{ifold}\ f\ g\ z\ \mathsf{Emp} = z \\ \quad \mathsf{ifold}\ f\ g\ z\ (\mathsf{Ins}\ a\ A) = \text{if}\ \mathsf{mem}\ a\ A\ \text{then}\ \mathsf{ifold}\ f\ g\ z\ A\ \text{else}\ f\ (g\ a)\ (\mathsf{ifold}\ f\ g\ z\ A)\end{array}$$

However, our package does not yet support generalized recursion (i.e. allowing the recursive clause to contain not only references to the recursive result $\mathsf{ifold}\ f\ g\ z\ A$, but to the argument itself too, as in $\mathsf{mem}\ a\ A$), so we have to take a slight detour, first defining

$$\begin{array}{l}\texttt{nonfreerec}\ \mathsf{ifold}' : (\alpha \to \beta) \to \beta \to \alpha\ \mathsf{fset} \to \alpha\ \mathsf{fset} \times \beta\ \texttt{where} \\ \qquad \mathsf{ifold}'\ g\ z\ \mathsf{Emp} = (\mathsf{Emp}, z) \\ \quad \mathsf{ifold}'\ g\ z\ (\mathsf{Ins}\ a\ A) = \text{case}\ \mathsf{ifold}'\ g\ z\ A\ \text{of}\ (A', b) \Rightarrow \\ \qquad\qquad\qquad \text{if}\ \mathsf{mem}\ a\ A\ \text{then}\ (A', b)\ \text{else}\ (A', f\ (g\ a)\ b)\end{array}$$

and then defining ifold $f\ g\ z = \mathsf{snd} \circ \mathsf{ifold}'\ f\ g\ z$. (This is the standard way to encode generalized recursion as primitive recursion.) The same trick can also define the HOL4 and PVS iterators, which assume a fixed left-commutative $f : \alpha \to \beta \to \beta$:

$$\begin{aligned}&\texttt{nonfreerec}\ \mathsf{hfold} : \beta \to \alpha\ \mathsf{fset} \to \beta\ \texttt{where}\\ &\qquad \mathsf{hfold}\ z\ \mathsf{Emp} \;=\; z\\ &\qquad \mathsf{hfold}\ (\mathsf{Ins}\ a\ A) \;=\; \text{if}\ \mathsf{mem}\ a\ A\ \text{then}\ \mathsf{hfold}\ z\ A\ \text{else}\ f\ a\ (\mathsf{hfold}\ z\ A)\end{aligned}$$

In the HOL4 standard library, hfold operates on the larger type α set; Homeier [11] also defines a custom finite-set type and defines hfold on it using his quotient package. In PVS, hfold operates on a custom subtype of α set. Isabelle also has an iterator customized to nonempty finite sets, definable by nonfree recursion (see below).

The benefit of all these representations is that each offers a different view, hence a different recursion principle for (nonempty) finite sets. Tannen and Subrahmanyam [25] argue with examples that each view has its own advantage, and also relates some of these views uniformly for sets, bags and lists in a categorical framework. In order to really take advantage of these offers, we of course need to pick one single type where we transfer all these principles. It turns out that proving embedding or isomorphisms between these types is relatively easy: all are defined using nonfree recursion. For instance,

$$\begin{aligned}&\texttt{nonfreerec}\ \mathsf{K} : \alpha\ \mathsf{fset} \to \alpha\ \mathsf{fset}'\ \texttt{where}\\ &\qquad \mathsf{K}\ \mathsf{Emp} \;=\; \mathsf{Emp}'\\ &\qquad \mathsf{K}\ (\mathsf{Ins}\ a\ A) \;=\; \mathsf{Un}\ (\mathsf{Singl}\ a)\ (\mathsf{K}\ A)\end{aligned}$$

establishing a bijection between the two types, as can be relatively easily checked (after proving a few lemmas). Since finite sets are developed in Isabelle using the predicate finite on the type α set of arbitrary sets, we actually prefer to transport all the recursion principles there, by embedding the algebraic types. After the isomorphic transfer to α set, we enable a variety of iterators restricted to finite sets, each having its on own contract.

Another view to nonempty finite sets we considered is the following singleton-insert view:

$$\begin{aligned}&\texttt{datatype}\ \alpha\ \mathsf{nfset}' \;=\; \mathsf{Singl}\ \alpha \mid \mathsf{Ins}\ \alpha\ (\alpha\ \mathsf{nfset}')\\ &\texttt{where}\ \mathsf{Ins}_1\texttt{:}\ \mathsf{Ins}\ a\ (\mathsf{Ins}\ b\ A) \;=\; \mathsf{Ins}\ b\ (\mathsf{Ins}\ a\ A)\\ &\quad \texttt{and}\ \mathsf{Ins}_2\texttt{:}\ \mathsf{Ins}\ a\ (\mathsf{Ins}\ a\ A) \;=\; \mathsf{Ins}\ a\ A\end{aligned}$$

In this view, we can define Nipkow and Paulson's recursor for finite sets [17], where an associative-commutative operator $f : \alpha \to \alpha \to \alpha$ is fixed:

$$\begin{aligned}&\texttt{nonfreerec}\ \mathsf{nfold} : \alpha\ \mathsf{fset} \to \alpha\ \texttt{where}\\ &\qquad \mathsf{nfold}\ (\mathsf{Singl}\ a) \;=\; a\\ &\qquad \mathsf{nfold}\ (\mathsf{Ins}\ a\ A) \;=\; \text{if}\ \mathsf{mem}\ a\ A\ \text{then}\ \mathsf{nfold}\ A\ \text{else}\ f\ a\ (\mathsf{nfold}\ A)\end{aligned}$$

Here is an example of an embedding into the bigger type α fset. For instance, the following embeds α fset$'$ into α set:

$$\begin{aligned} &\texttt{nonfreerec}\ \mathsf{J} : \alpha\ \mathsf{fset}' \to \alpha\ \mathsf{set}\ \texttt{where} \\ &\mathsf{J}\ \mathsf{Emp}' = \emptyset \\ &\mathsf{J}\ (\mathsf{Singl}\ a) = \{a\} \\ &\mathsf{J}\ (\mathsf{Un}\ A_1\ A_2) = \mathsf{J}\ A_1 \cup \mathsf{J}\ A_2 \end{aligned}$$

That J is an injection and that its image consists of precisely the finite sets follow routinely by fset$'$-induction and finite-induction, respectively.

The algebraic specifications of finite (nonempty) sets yield the following collection of iterators:

- fold_fset : $\beta \to (\alpha \to \beta \to \beta) \to \beta \to \alpha$ set,
- fold_fset$'$: $\beta \to (\alpha \to \beta) \to (\beta \to \beta \to \beta) \to \beta \to \alpha$ set,
- fold_nfset : $\beta \to (\beta \to \beta \to \beta) \to \beta \to \alpha$ set,
- fold_nfset$'$: $(\alpha \to \beta) \to (\alpha \to \beta \to \beta) \to \beta \to \alpha$ set,

each with its own contract:

- If fset_alg $E\ I$, then:
  - fold_fset $E\ I$ Emp $= E$
  - $\forall a\ A$. finite $A \longrightarrow$ fold_fset $E\ I\ (\{a\} \cup A) = I\ a$ (fold_fset $E\ I\ A$)
- If fset_alg$'$ $E\ S\ U$, then:
  - fold_fset$'$ $E\ S\ U$ Emp $= E$
  - $\forall a$. fold_fset$'$ $E\ S\ U\ \{a\} = S\ a$
  - $\forall A_1\ A_2$. finite $A_1 \wedge$ finite $A_2 \longrightarrow$ fold_fset$'$ $E\ S\ U\ (A_1 \cup A_2) = U$ (fold_fset$'$ $E\ S\ U\ A_1$) (fold_fset$'$ $E\ S\ U\ A_2$)
- If nfset_alg $S\ U$, then:
  - $\forall a$. fold_nfset $S\ U\ \{a\} = S\ a$
  - $\forall A_1\ A_2$. finite $A_1 \wedge$ finite $A_2 \wedge \emptyset \notin \{A_1, A_2\} \longrightarrow$ fold_nfset $S\ U\ (A_1 \cup A_2) = U$ (fold_nfset $S\ U\ A_1$) (fold_nfset $S\ U\ A_2$)
- If fset_alg $S\ I$, then:
  - fold_fset $S\ I$ Emp $= E$
  - $\forall a\ A$. finite $A \wedge A \neq \emptyset \longrightarrow$ fold_fset $S\ I\ (\{a\} \cup A) = I\ a$ (fold_fset $S\ I\ A$)

where fset_alg, fset_alg$'$ and nfset_alg are the predicates corresponding to the various algebraic-datatype specifications (analogous to bag_alg):

- fset_alg $E\ I$—I is left-commutative and left-idempotent with unit E;
- fset_alg$'$ $E\ S\ U$—U is ACI with unit E;
- nfset_alg $S\ U$—U is ACI;
- nfset_alg$'$ $S\ I$—I is left-commutative and left-idempotent.

B.2 Sum of Two Algebraic Structures

A construction similar to the one for polynomials in §2.2 yields the sum of two rings, or any other algebraic structures, such as semigroups:

$$\begin{array}{l}\texttt{datatype}\ (\alpha : \mathsf{semigroup}, \beta : \mathsf{semigroup})\ \mathsf{sum} \ = \ \mathsf{Left}\ \alpha \mid \mathsf{Right}\,\beta \mid \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{Times}\ ((\alpha,\beta)\ \mathsf{sum})\ ((\alpha,\beta)\ \mathsf{sum}) \\ \texttt{where}\ a_1 * a_2 = a \ \longrightarrow\ \mathsf{Times}\ (\mathsf{Left}\ a_1)\ (\mathsf{Left}\ a_2) = \mathsf{Left}\ a \\ \quad \texttt{and}\ a_1 * a_2 = a \ \longrightarrow\ \mathsf{Times}\ (\mathsf{Right}\ a_1)\ (\mathsf{Right}\ a_2) = \mathsf{Right}\ a \\ \quad \texttt{and}\ \mathsf{Times}\ (\mathsf{Times}\ x\ y)\ z \ = \ \mathsf{Times}\ x\ (\mathsf{Times}\ y\ z)\end{array}$$

In general, for a class of algebras $\mathscr{A}$ and $\mathscr{B}$ specified equationally (such as groups, rings, etc.), their sum is built by taking the free algebra of terms over the set-theoretic sum of their carriers $A + B$, factored to the following equations:

- identification of the new operation with the already existing ones on the $\mathscr{A}$ and $\mathscr{B}$ fragments (e.g., the first two clauses above);
- the equations characterizing the class of algebras (e.g., the third equation above).

B.3 Higher Order Abstract Syntax

In the context of §2.3, we define the higher-order abstract syntax (HOAS interpretation of λ-terms into λ-terms over constants augmented with constants for application and abstraction:

$$\texttt{datatype}\ \beta\,\mathsf{const} \ = \ \mathsf{Old}\ \beta \mid \mathsf{app} \mid \mathsf{lam}$$

The definition includes a statement of substitution and freshness compositionality:

$$\begin{array}{rcl}\texttt{nonfreerec}\ \mathsf{hoas} : (\alpha,\beta)\ \mathsf{lterm} \to (\alpha, \beta\,\mathsf{const})\ \mathsf{lterm}\ \texttt{where} & & \\ \mathsf{hoas}\ (\mathsf{Ct}\ c) & = & \mathsf{Ct}\ (\mathsf{Old}\ c) \\ \mathsf{hoas}\ (\mathsf{Var}\ y) & = & \mathsf{Var}\ y \\ \mathsf{hoas}\ (\mathsf{App}\ s\ t) & = & \mathsf{App}\ (\mathsf{App}\ \mathsf{app}\ (\mathsf{hoas}\ s))\ (\mathsf{hoas}\ t) \\ \mathsf{hoas}\ (\mathsf{Lam}\ x\ s) & = & \mathsf{Lam}\ x\ (\mathsf{hoas}\ s) \\ \mathsf{hoas}\ (s\ [t/y]) & = & (\mathsf{hoas}\ s)\ [(\mathsf{hoas}\ t)\ /\ y] \\ \mathsf{fresh}\ y\ s & \longrightarrow & \mathsf{fresh}\ y\ (\mathsf{hoas}\ s)\end{array}$$

B.4 Regular Expressions

The following defines regular expressions over an alphabet α up to the Kleene algebra equations, known to form a complete axiomatization of equality up to the generated language.

$$\begin{array}{l}\texttt{datatype}\ \alpha\,\mathsf{reg} \ = \ \mathsf{Let}\ \alpha \mid \mathsf{Zero} \mid \mathsf{One} \mid \\ \qquad\qquad\qquad\qquad \mathsf{Plus}\ (\alpha\ \mathsf{fset}')\ (\alpha\ \mathsf{fset}') \mid \mathsf{Times}\ (\alpha\ \mathsf{fset}')\ (\alpha\ \mathsf{fset}') \\ \texttt{where}\ (e_1 + e_2) + e_3 \ = \ e_1 + (e_2 + e_3) \quad \texttt{and} \quad e_1 + e_2 \ = \ e_2 + e_3 \\ \quad \texttt{and}\ 0 + e \ = \ e \quad \texttt{and} \quad e + e = e \quad \texttt{and} \quad (e_1 \cdot e_2) \cdot e_3 \ = \ e_1 \cdot (e_2 \cdot e_3) \\ \quad \texttt{and}\ 1 \cdot e \ = \ e \quad \texttt{and} \quad e \cdot 1 \ = \ e \quad \texttt{and} \quad 0 \cdot e \ = \ 0 \\ \quad \texttt{and}\ e_1 \cdot (e_2 + e_3) \ = \ (e_1 \cdot e_2) + (e_1 \cdot e_3) \\ \quad \texttt{and}\ (1 + (e^* \cdot e)) + e^* = e^* \quad \texttt{and} \quad (1 + (e \cdot e^*)) + e^* = e^* \\ \quad \texttt{and}\ (e \cdot e_1) + e_1 = e_1 \ \longrightarrow\ (e^* \cdot e_1) + e_1 = e_1 \\ \quad \texttt{and}\ (e_1 \cdot e) + e_1 = e_1 \ \longrightarrow\ (e_1 \cdot e^*) + e_1 = e_1\end{array}$$

where we wrote infix $+$ and $\times$ for Plus and Times and $\cdot$, postfix * for Star, and 0 and 1 for Zero and One.[5] The last two equations are conditional; they express the minimality of e^* among the prefix points of $\lambda x.\ x \cdot e$ and $\lambda x.\ e \cdot x$ w.r.t. the order $a \leq b$ defined as $a + b = b$.

The generated-language operator can then be defined by nonfree recursion using the same clauses as for standard recursion:

$$\begin{aligned}
&\texttt{nonfreerec}\ \mathsf{lang} : \alpha\ \mathsf{reg} \to (\alpha\ \mathsf{list})\ \mathsf{set}\ \texttt{where}\\
&\qquad \mathsf{lang}\ (\mathsf{Let}\ a) \ = \ \{[a]\}\\
&\qquad \mathsf{lang}\ (e_1 \cdot e_2) \ = \ \{w_1 @ w_2 \mid w_1 \in \mathsf{lang}\ e_1 \ \wedge\ \mathsf{lang}\ e_2\}\\
&\qquad \mathsf{lang}\ e^* \ = \ \mathsf{Kl}\ (\mathsf{lang}\ e)\\
&\qquad\quad *** \ \text{etc} \ ***
\end{aligned}$$

Where Kl is the Kleene-star operator on languages. So can be the standard interpretation in the algebra of relations over a type β, parameterized by an interpretation of the letters:

$$\begin{aligned}
&\texttt{nonfreerec}\ \mathsf{rint} : (\alpha \to (\beta \times \beta)\ \mathsf{set}) \to \alpha\ \mathsf{reg} \to (\beta \times \beta)\ \mathsf{set})\ \texttt{where}\\
&\qquad \mathsf{rint}\ f\ (\mathsf{Let}\ a) \ = \ f\ a\\
&\qquad \mathsf{rint}\ f\ (e_1 \cdot e_2) \ = \ \mathsf{rint}\ f\ e_1 \ \circ\ \mathsf{rint}\ f\ e_2\\
&\qquad \mathsf{rint}\ f\ e^* \ = \ \mathsf{Tr}\ (\mathsf{rint}\ f\ e)\\
&\qquad\quad *** \ \text{etc} \ ***
\end{aligned}$$

where Tr is the reflexive-transitive closure on relations. In both cases, the emerging goals are proved using library facts about lists, sets and relations.

C Algorithmic Rule Systems

The various transformations on terms that are necessary to animate the metatheory are realized as algorithmic rule systems. These are collections of rules about moded judgments, which are defined predicates in Isabelle/HOL. The definition of such a judgment constitutes its semantic meaning, while the rules are theorems that constitute the algorithm we use to synthesize the outputs and establish the judgment.

The animation strategy for algorithmic rule systems can be regarded as a deterministic variant of Lambda-Prolog [16], with a local backtracking primitive, support for non-pattern matching and some Isabelle-specific enhancements. At the moment we provide no mechanisms for checking totality of these algorithms, which justify meta arguments about such rule systems in Twelf [20]. Such a justification is not pressing in our case because we only animate the rules on ground terms, but additional checks would be beneficial of course.

The main benefit over tactics is the ease of development because higher-order term manipulation is direct, the soundness of the composition of facts is separated from the

[5] Here and elsewhere: We use infix notations for the readability of this paper—currently the package does not provide support for it.

dynamic invocation and is proved only once, treatment of contexts is implicit, modifications of the various judgments just consists in emitting clauses (esp. local ones as premises of subgoals) instead of tinkering with ML data structures.

Forward Rules Integrated with the infrastructure for algorithmic rule systems is a novel algorithmic principle for context enrichment, which we call "forward rules". They drive the animation of the metatheory that we describe below.

For reasons of space we only give a conceptual motivation. We want to declaratively specify the generation of theory content, i.e. definitions of type constructors, terms and theorems. Forward rules are small theory extension steps represented as implicational theorems about moded judgments. Algorithmically speaking, they wait for input facts, process them with algorithmic rule systems, issue foundational theory extension mechanisms as side effects, and make output facts available.

C.1 Simplified Isomorphic Transfer

Isomorphic Transfer refers to an algorithmic rule system that synthesizes isomorphisms away from an input setoid and transforms input terms to their image under the isomorphism away from the input setoid. We do not show this rule system in full generality here, but rather focus on a simplified version on types instead of setoids. For more details we refer to the first author's master thesis [22].

We need the notion of lifting bijections $f : \alpha \to \alpha'$, $g : \beta \to \beta'$ to the function space:

$$f \gg g \;:=\; (\lambda h : \alpha \to \beta.\, g \circ h \circ f^{-1}) : (\alpha \to \beta) \to (\alpha' \to \beta')$$

We can then specify how to form isomorphisms away from an input type. This is realized by the following rules for the judgment $\alpha \simeq_{\blacktriangleright} \alpha'$ via $f \;:=\; \mathsf{bijection}\ f : \alpha \to \alpha'$ with input α.

$$\frac{\alpha \simeq_{\blacktriangleright} \alpha' \text{ via } f \qquad \beta \simeq_{\blacktriangleright} \beta' \text{ via } g}{(\alpha \to \beta) \simeq_{\blacktriangleright} (\alpha' \to \beta') \text{ via } f \gg g} \qquad \frac{}{\mathsf{bool} \simeq_{\blacktriangleright} \mathsf{bool} \text{ via } \mathsf{id}}$$

We use the identity on booleans because we want to map true propositions to true propositions. Rules for other type constructors are formulated similarly.

An isomorphism away from an input type has to map term constructions that inhabit the type to corresponding concepts in the target type. The term transformation is represented by the judgment $t : \alpha \mapsto t' : \alpha'$ via $f \;:=\; \mathsf{bijection}\ f : \alpha \to \alpha' \wedge f\ t = t'$ with input t. We now give rules for transforming selected term constructions.

$$\frac{t_1 : (\alpha \to \beta) \mapsto t_1' : (\alpha' \to \beta') \text{ via } f \gg g \qquad t_2 : \alpha \mapsto t_2' : \alpha' \text{ via } f}{(t_1\ t_2) : \beta \mapsto (t_1'\ t_2') : \beta' \text{ via } g}$$

$$\frac{\alpha \simeq_{\blacktriangleright} \alpha' \text{ via } f \qquad \forall x, x'.\quad x : \alpha \mapsto x' : \alpha' \text{ via } f \quad\to\quad (t\ x) : \beta \mapsto (t'\ x') : \beta' \text{ via } g}{(\lambda x : \alpha.\, t\ x) : (\alpha \to \beta) \mapsto (\lambda x' : \alpha'.\, t'\ x') : (\alpha' \to \beta') \text{ via } f \gg g}$$

$$\frac{\alpha \simeq_{\blacktriangleright} \alpha' \text{ via } f}{(\forall) : ((\alpha \to \mathsf{bool}) \to \mathsf{bool}) \mapsto (\forall) : ((\alpha' \to \mathsf{bool}) \to \mathsf{bool}) \text{ via } (f \gg \mathsf{id}) \gg \mathsf{id}}$$

The rules for the other logical constants are similar.

The algorithmic meaning of the abstraction rule is the following: to transform an abstraction $(\lambda x : \alpha.\, t\, x)$, we synthesize the isomorphism f away from the domain α, to α' (synthesized), with a subcall to the judgment $\alpha \simeq_{\rightarrow} \alpha'$ via f. We then transform the body of the abstraction. We fix fresh variables x, x', register the local rule $x : \alpha \mapsto x' : \alpha'$ via f stating that x maps to x' under the isomorphism f and start the recursive transformation of the body $t\ x$. This recursive call results in output types and terms that we match against $\beta, \beta', t'\ x'$ (t' is the unknown, x' is fixed) and g. Matching against $t'\ x'$ works by higher-order pattern matching. Finally we construct the outputs $(\lambda x' : \alpha'.\, t'\ x')$, $\alpha' \to \beta'$, $f \gg g$ of the judgment and compose the transformation rule with the theorems stating instances of its premises, to derive its conclusion.

To invoke the isomorphic transfer under a user-specified isomorphism, we register additional rules that describe the construction of isomorphisms away from the additional type constructions and provide rules that realize the transfer of the elementary terms in these types.

Because of the tendency to employ under-specifications in HOL, we actually implemented the transfer between setoids, not just types. Noteworthy is also the algorithmic isomorphism that realizes currying on finite products, which is ad hoc higher-order term transformation that overrides the default behaviour of the application rule and is prominently used when animating the metatheory.

D More Details on The Instantiation of the Metatheory

We give a high-level overview of the steps necessary to animate the metatheory :

- We introduce finite datatypes to instantiate the sorts, parameter sorts, operation and relation symbols of the signature.
- We construct functions on these finite datatypes, representing the sorting of operations and relations
- We check inhabitedness of the sorts by forward chaining of sort inhabitedness implications resulting from the sorting of operations.
- We embed the parameter types into their finite sum type, the parameter universe, and register bijections out of these subsets back to the parameter types. We use the inverse of this isomorphism to transfer the parameter conditions, as given by the user, to relations over the parameter universe.
- We typecheck the Horn formulas and reflect them into the representation used by the metatheory. Parameter conditions become the corresponding relations over the parameter universe. We establish the well-sortedness of these Horn formulas.
- We instantiate the locale which contains the metatheory and discharge its assumptions with the facts established with the steps above.
- For each sort, we emit a typedef with the quotient terms of this sort as the representing set. These newly defined types are the mutually recursive datatypes we present to the user. We register bijections from the representing sets to them.
- We transform the term interpretation of the operation and relation symbols under this isomorphism. We define the constructors and relations on the datatypes by emitting definitions with these transformation results.

– We transfer and post-process the meta-theorems to propositions about the mutually recursive datatypes.

The last step, i.e. the animation of the meta-theorems, employs mainly the following transformation steps:

– Quantifications over sorts, parameters sorts, operation and relation symbols are unrolled to become conjunctions, which is just a rewriting step.
– Quantification over a variable interpretation becomes quantifications over single variable values. We achieve this by updating the variable interpretation on all occurring variables to use universally quantified values and evaluating the definition of satisfiability. This gets rid of the variable interpretation in favor of the universally quantified values.
– Quantification over a sort-dependent predicate becomes quantifications over predicates, one for each sort. We achieve this by rewriting bounded quantifications over sort-dependent predicates.
– Quantification over tuples in finite products becomes quantifications over elements, with the tuple variables replaced by tuples containing the element variables.
– We transfer under the isomorphism from the term model and the parameter universe to the types presented to the user.
– We apply standard rewrite rules to achieve palatable propositions adhering to Isabelle's rule format.

We will show the transformation of the induction meta-theorem of the λ-terms example. We include currying here, which was omitted in section 5.2. The recursive function $\mathsf{product} : \alpha\ \mathsf{set}\ \mathsf{list} \to \alpha\ \mathsf{list}\ \mathsf{set}$, $\mathsf{product}\ \mathsf{Nil} := \{[]\}$, $\mathsf{product}\ (\mathsf{Cons}\ A\ As) := \{\mathsf{Cons}\ x\ xs \mid x \in A,\ xs \in \mathsf{product}\ As\}$ is necessary here to state bounded quantifications. To ease the presentation we write types instead of UNIV sets over them. Due to reasons of space we will omit uninteresting parts of the proposition, writing … instead. We start from a minor reformulation of the meta-theorem:

$$\begin{aligned}
&\forall s.\ \forall \varphi \in (\Pi s2.\ (\mathsf{htrms}\ s2 \to \mathsf{bool})).\ \forall H \in \mathsf{htrms}\ s.\\
&\quad (\forall \sigma.\ \forall pl \in \mathsf{product}\ (\mathsf{map}\ \mathsf{params}\ (\mathsf{arOfP}\ \sigma)).\ \forall Hl \in \mathsf{product}\ (\mathsf{map}\ \mathsf{htrms}\ (\mathsf{arOf}\ \sigma)).\\
&\qquad \mathsf{list_all2}\ \varphi\ (\mathsf{arOf}\ \sigma)\ Hl \to \varphi\ (\mathsf{stOf}\ \sigma)\ (\mathsf{Hop}\ \sigma\ pl\ Hl))\\
&\to \varphi\ s\ H.
\end{aligned}$$

First we unroll the sort quantifications. In our case there is only one sort, lt. We unfold the quantification over operation symbols σ, omitting parts of the proposition corresponding to $\sigma = \mathsf{ct}$, $\sigma = \mathsf{subst}$. We unfold the definition of arOfP, arOf, stOf and the

rewrite with the computational rules for map.

$$\begin{array}{l}
\forall \varphi \in (\Pi s2.\ (\mathsf{htrms}\ s2 \to \mathsf{bool})).\ \forall H \in \mathsf{htrms\ lt}. \\
\quad (\forall pl \in \mathsf{product}\ [\mathsf{params\ a}].\ \forall Hl \in \mathsf{product}\ []. \\
\qquad \mathsf{list_all2}\ \varphi\ []\ Hl \to \varphi\ \mathsf{lt}\ (\mathsf{Hop\ var}\ pl\ Hl)) \\
\wedge\ (\forall pl \in \mathsf{product}\ []. \forall Hl \in \mathsf{product}\ [\mathsf{htrms\ lt},\ \mathsf{htrms\ lt}]. \\
\qquad \mathsf{list_all2}\ \varphi\ [\mathsf{lt}, \mathsf{lt}]\ Hl \to \varphi\ \mathsf{lt}\ (\mathsf{Hop\ app}\ pl\ Hl)) \\
\wedge\ (\forall pl \in \mathsf{product}\ [\mathsf{params\ a}]. \forall Hl \in \mathsf{product}\ [\mathsf{htrms\ lt}]. \\
\qquad \mathsf{list_all2}\ \varphi\ [\mathsf{lt}]\ Hl \to \varphi\ \mathsf{lt}\ (\mathsf{Hop\ lam}\ pl\ Hl)) \\
\wedge\ \ldots \qquad \to \qquad \varphi\ \mathsf{lt}\ H.
\end{array}$$

Now we unroll the quantifications over finite products pl, Hl and sort-dependent predicates φ and simplify further with the computational rule for list_all2.

$$\begin{array}{l}
\forall \varphi_0 \in (\mathsf{htrms\ lt} \to \mathsf{bool}).\ \forall H \in \mathsf{htrms\ lt}. \\
\quad (\forall p_1 \in \mathsf{params\ a}.\quad \varphi_0\ (\mathsf{Hop\ var}\ [p_1]\ [])) \\
\wedge\ (\forall H_1 \in \mathsf{htrms\ lt}.\ \forall H_2 \in \mathsf{htrms\ lt}.\quad \varphi_0\ H_1 \wedge \varphi_0\ H_2 \to \varphi_0\ (\mathsf{Hop\ app}\ []\ [H_1, H_2])) \\
\wedge\ (\forall p_1 \in \mathsf{params\ a}.\ \forall H_1 \in \mathsf{htrms\ lt}.\quad \varphi_0\ H_1 \to \varphi_0\ (\mathsf{Hop\ lam}\ [p_1]\ [H_1])) \\
\wedge\ \ldots \qquad \to \qquad \varphi_0\ H.
\end{array}$$

Now we transfer under the isomorphisms from the quotient term model to the newly-defined datatype and from the parameter universe to the parameter type: $\mathsf{htrms\ lt} \to \alpha\ \mathsf{lterm}$, $\mathsf{params\ a} \to \alpha$. This results in:

$$\begin{array}{l}
\forall \varphi_0 \in (\alpha\ \mathsf{lterm} \to \mathsf{bool}).\ \forall H \in \alpha\ \mathsf{lterm}. \\
\quad (\forall p_1 \in \alpha.\quad \varphi_0\ (\mathsf{Var}\ p_1)) \\
\wedge\ (\forall H_1 \in \alpha\ \mathsf{lterm}.\ \forall H_2 \in \alpha\ \mathsf{lterm}.\quad \varphi_0\ H_1 \wedge \varphi_0\ H_2 \to \varphi_0\ (\mathsf{App}\ H_1\ H_2)) \\
\wedge\ (\forall p_1 \in \alpha.\ \forall H_1 \in \alpha\ \mathsf{lterm}.\quad \varphi_0\ H_1 \to \varphi_0\ (\mathsf{Lam}\ p_1\ H_1)) \\
\wedge\ \ldots \qquad \to \qquad \varphi_0\ H.
\end{array}$$

We go on to apply further standard simplification rules to make this result adhere to Isabelle's rule format.

D.1 Animating the Iteration Principle

The animation of the iteration principle needs to construct the target interpretation of sorts, operation and relation symbols first. This is analogous to the construction of the parameter sort and parameter condition interpretations, so we omit a discussion. Satisfaction of the Horn clauses in this interpretation is shown by processing the satisfaction of each Horn clause under this interpretation with our transformation steps and comparing the result with the user-proven facts.

The generic iterator applied to each sort is transformed under the isomorphism into a family of iterators, one for each of the mutually recursive datatypes. The iteration principle and the preservation of relational consequences under iteration is also transformed with our steps above.