

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:

$$\begin{aligned} \text{datatype } \alpha \text{ bag} &= \text{BEmp} \mid \text{Blns } \alpha (\alpha \text{ bag}) \\ \text{where } \text{Blns } a_1 (\text{Blns } a_2 B) &= \text{Blns } a_2 (\text{Blns } a_1 B) \end{aligned}$$

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 [23] and theorem proving. However, mainstream proof assistants based on type theory [1, 5] or higher-order logic (HOL) [10, 17] currently do not provide mechanisms for specifying nonfree datatypes directly.

In HOL-based provers, such as our favorite one Isabelle/HOL [17], 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 with 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 package is compatible with Isabelle2013 and is publicly available [21].

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 [17] 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., α list and α set denote the list and powerset types over α . Polymorphic types are not syntactically distinguished—e.g., α list also denotes the polymorphic type $\forall \alpha. \alpha$ list. We write $\alpha \rightarrow \beta$, $\alpha + \beta$, 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 $\alpha \rightarrow \beta$ 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., $\tau : \text{semigroup}$ means that there exists an operation $+$: $\tau \rightarrow \tau \rightarrow \tau$ 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 $\text{BEmp} : \alpha \text{ bag}$ and $\text{Blns} : \alpha \rightarrow \alpha \text{ bag} \rightarrow \alpha \text{ bag}$,
- the iterator $\text{iter_bag} : \beta \rightarrow (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \alpha \text{ bag} \rightarrow \beta$.

In addition, several characteristic theorems are derived. They include facts also available for standard free datatypes:

- Case distinction: $(B = \text{BEmp} \rightarrow \varphi) \wedge (\forall a C. B = \text{Blns } a C \rightarrow \varphi) \rightarrow \varphi$
- Induction: $\varphi \text{ BEmp} \wedge (\forall a B. \varphi B \rightarrow \varphi (\text{Blns } a B)) \rightarrow (\forall B. \varphi B)$

Note that the injectivity of the constructors, here,

$$\text{Blns } a_1 B_1 = \text{Blns } a_2 B_2 \rightarrow a_1 = a_2 \wedge B_1 = B_2,$$

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:
 $\text{Blns } a_1 (\text{Blns } a_2 B) = \text{Blns } a_2 (\text{Blns } a_1 B)$
- The recursion principle, consisting of conditional equations for iteration:
 $\text{bag_alg } E I \rightarrow \text{iter_bag } E I \text{ BEmp} = E$
 $\text{bag_alg } E I \rightarrow (\forall a B. \text{iter_bag } E I (\text{Blns } a B) = I a (\text{iter_bag } E I B))$

where $\text{bag_alg } E I$ 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 $(\beta, E : \beta, I : \alpha \rightarrow \beta \rightarrow \beta)$ satisfying the equation (with E and I replacing BEmp and Blns) as expressed by the predicate $\text{bag_alg } E I$. 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 $\text{iter_bag } E I$. Its uniqueness is given by the induction principle.

As with other definitional packages for recursion, the user does not need 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:

$$\begin{aligned} \text{nonfreerec bag_map} : (\alpha \rightarrow \beta) \rightarrow \alpha \text{ bag} \rightarrow \beta \text{ bag} \text{ where} \\ \text{bag_map } f \text{ BEmp} &= \text{BEmp} \\ \text{bag_map } f (\text{Blns } a B) &= \text{Blns } (f a) (\text{bag_map } f B) \end{aligned}$$

In response to this command, the package does the following (for a fixed $f : \alpha \rightarrow \beta$):

- (1) identifies E and I as being $\text{BEmp} : \beta \text{ bag}$ and $(\lambda a. \text{Blns } (f a)) : \alpha \rightarrow \beta \text{ bag} \rightarrow \beta \text{ bag}$;
- (2) defines $\text{bag_map } f = \text{iter_bag } E I$;
- (3) prompts the user to discharge the goal $\text{bag_alg } E I$;
- (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,

$$\text{Blns } (f a_1) (\text{Blns } (f a_2) B) = \text{Blns } (f a_2) (\text{Blns } (f a_1) B)$$

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:

$$\begin{aligned} \text{nonfreerec sum} &: (\alpha \rightarrow \text{nat}) \rightarrow \alpha \text{ bag} \rightarrow \text{nat} \text{ where} \\ \text{sum } f \text{ BEmp} &= 0 \\ \text{sum } f (\text{Blns } a B) &= \text{sum } f B + f a \end{aligned}$$

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 $\beta : \text{comm_monoid_add}$, covering all types equipped with a commutative monoid structure $(\beta, 0, +)$. The multiplicity of an element in a bag, `mult` : $\alpha \rightarrow \alpha \text{ bag} \rightarrow \text{nat}$ is obtained as `mult a B = sum ($\lambda a'$. if $a = a'$ then 1 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.

```
datatype ( $\alpha : \text{comm\_ring}$ ,  $\beta$ ) poly = Sc  $\alpha$  | Var  $\beta$  | Uminus (( $\alpha, \beta$ ) poly) |
  Plus (( $\alpha, \beta$ ) poly) (( $\alpha, \beta$ ) poly) | Times (( $\alpha, \beta$ ) poly) (( $\alpha, \beta$ ) poly)
where ( $-a_1$ ) =  $a \rightarrow$  Uminus (Sc  $a_1$ ) = Sc  $a$ 
and  $a_1 + a_2 = a \rightarrow$  Plus (Sc  $a_1$ ) (Sc  $a_2$ ) = Sc  $a$ 
and  $a_1 * a_2 = a \rightarrow$  Times (Sc  $a_1$ ) (Sc  $a_2$ ) = Sc  $a$ 
and Plus (Sc 0)  $P = P$ 
and Plus (Plus  $P_1 P_2$ )  $P_3 =$  Plus  $P_1$  (Plus  $P_2 P_3$ )
*** Etc.: All the commutative-ring axioms for Plus, Times, Sc 0 ***
```

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 parameter variables a_1, a_2, a from type α . Strictly speaking, each of the clauses forms an infinite family of Horn clauses, indexed by $a, a_1, a_2 : \alpha$. 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 $f : (\alpha : \text{comm_ring}) \rightarrow (\gamma : \text{comm_ring})$ (assumed to commute with $+$, $*$, 0) and variable interpretations $g : \beta \rightarrow \gamma$, to morphisms $\text{ext } f \ g : (\alpha, \beta) \text{ poly} \rightarrow \gamma$. In the context of such f and g , we define ext by simply writing down its desired interaction with the polynomial operators:

$$\begin{aligned} & \text{nonfreerec ext} : (\alpha, \beta) \text{ poly} \rightarrow \gamma \text{ where} \\ \text{ext (Sc } a) &= f \ a & \text{ext (Var } b) &= g \ b & \text{ext (Uminus } P) &= - \text{ext } P \\ \text{ext (Plus } P \ Q) &= \text{ext } P + \text{ext } Q & \text{ext (Times } P \ Q) &= \text{ext } P * \text{ext } Q \end{aligned}$$

where simplification with the ring axioms of γ and the morphism axioms of f immediately discharges the goals (resulting from the `nonfreerec` command). Polynomial evaluation is obtained from ext taking $f = \text{id}$.

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 [18, 19]. 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:

```
datatype ( $\alpha, \beta$ ) lterm = Var  $\alpha$  | Ct  $\beta$  | App (( $\alpha, \beta$ ) lterm) (( $\alpha, \beta$ ) lterm) |
  Lam  $\alpha$  (( $\alpha, \beta$ ) lterm) | Subst (( $\alpha, \beta$ ) lterm) (( $\alpha, \beta$ ) lterm)  $\alpha$ 
```

with `fresh : $\alpha \rightarrow (\alpha, \beta)$ lterm \rightarrow bool`

```
where (Var  $x$ ) [t/x] = t
  and  $x \neq y \rightarrow$  (Var  $y$ ) [t/x] = Var  $y$ 
  and (Ct  $c$ ) [t/x] = Ct  $c$ 
  and (App  $s_1$   $s_2$ ) [t/x] = App ( $s_1$  [t/x]) ( $s_2$  [t/x])
  and  $x \neq y \wedge$  fresh  $y$   $t \rightarrow$  (Lam  $y$   $s$ ) [t/x] = Lam  $y$  ( $s$  [t/x])
  and  $x \neq y \rightarrow$  fresh  $x$  (Var  $y$ )
  and fresh  $x$  (Ct  $c$ )
  and fresh  $x$   $s_1 \wedge$  fresh  $x$   $s_2 \rightarrow$  fresh  $x$  (App  $s_1$   $s_2$ )
  and fresh  $x$  (Lam  $x$   $s$ )
  and fresh  $x$   $s \rightarrow$  fresh  $x$  (Lam  $y$   $s$ )
  and  $x \neq y \wedge$  fresh  $x$   $s \rightarrow$  Lam  $y$   $s$  = Lam  $x$  ( $s$  [Var  $x$  /  $y$ ])
```

where we wrote $s[t/x]$ instead of `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 [19] 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, `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 quotiented w.r.t. α) together with indicating the correct behavior w.r.t. freshness and substitution:

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

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 $\alpha \rightarrow \text{nat}$, freshness is interpreted as $\lambda y \ s. \text{occs } s \ y = 0$. 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”:

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

(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.)

$$\text{inductive } \equiv : \alpha \text{ lterm}' \rightarrow \alpha \text{ lterm}' \rightarrow \text{bool} \text{ and } \text{fresh}' : \alpha \text{ lterm}' \rightarrow \text{bool}$$

where

*** One clause for each user-specified Horn clause: ***

$$(\text{Var}' \ x) [t/x] \equiv t$$

$$\text{and } x \neq y \longrightarrow (\text{Var}' \ y) [t/x] \equiv \text{Var}' \ y$$

*** etc. ***

*** The equivalence rules: ***

$$\text{and } s \equiv s \quad \text{and } s_1 \equiv s_2 \longrightarrow s_2 \equiv s_1 \quad \text{and } s_1 \equiv s_2 \wedge s_2 \equiv s_3 \longrightarrow s_1 \equiv s_3$$

*** A congruence rule for each user-specified constructor: ***

$$\text{and } s_1 \equiv t_1 \wedge s_2 \equiv t_2 \longrightarrow \text{App}' \ s_1 \ s_2 \equiv \text{App}' \ t_1 \ t_2$$

*** etc. ***

*** A preservation rule for each constructor-predicate combination: ***

$$\text{and } s_1 \equiv t_1 \wedge s_2 \equiv t_2 \wedge \text{fresh}' \ x \ (\text{App}' \ s_1 \ s_2) \longrightarrow \text{fresh}' \ x \ (\text{App}' \ t_1 \ t_2)$$

*** etc. ***

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

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, we fix an arbitrary type β with operations and relations on it and assume it satisfies the Horn clauses. (6) A function $f : \alpha$ lterm' $\rightarrow \beta$ 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 $\rightarrow \beta$ 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 locale, 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 a set of Horn clauses over the signature. An example instantiation is given in §5.1. Both terms and clauses are deeply embedded. Sorts represent relevant Isabelle types. A specific feature 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. $\text{Inl} : \alpha \rightarrow \alpha + \beta$ and $\text{Inr} : \beta \rightarrow \alpha + \beta$ are the left and right injections into the sum type, and $\text{isInl}, \text{isInr} : \alpha + \beta \rightarrow \text{bool}$ are their corresponding discriminators; namely, $\text{isInl } c$ holds iff c has the form $\text{Inl } a$ for some a , and $\text{isInr } c$ holds iff c has the form $\text{Inr } b$ for some b . $[]$ is the empty list, $[a_1, \dots, a_n]$ is the list of the n indicated elements. $\text{map} : (\alpha \rightarrow \beta) \rightarrow \alpha \text{ list} \rightarrow \beta \text{ list}$ is the standard list-map operator, and $\text{map2} : (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \alpha \text{ list} \rightarrow \beta \text{ list} \rightarrow \gamma \text{ list}$ is its binary counterpart, with $\text{map2 } f [a_1, \dots, a_n] [b_1, \dots, b_n] = [f a_1 b_1, \dots, f a_n b_n]$. Similarly, $\text{list_all} : (\alpha \rightarrow \text{bool}) \rightarrow \alpha \text{ list} \rightarrow \text{bool}$ is the universal quantifier over lists, with $\text{list_all } \varphi [a_1, \dots, a_n]$ meaning that φa_i holds for all i , and $\text{list_all2} : (\alpha \rightarrow \beta \rightarrow \text{bool}) \rightarrow \alpha \text{ list} \rightarrow \beta \text{ list} \rightarrow \text{bool}$ is its binary counterpart, with $\text{list_all2 } \varphi al bl$ meaning that al has the form $[a_1, \dots, a_n]$, bl has the form $[b_1, \dots, b_n]$, and $\varphi a_i b_i$ holds for all i . In particular $\text{list_all2 } \varphi al bl$ 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:

$$\text{datatype (sort, opsym) trm} = \text{Var sort var} \mid \text{Op opsym (pvar list) (((sort, opsym) trm) list)}$$

Thus a term T is either a sorted variable $\text{Var } s \ x$ or has the form $\text{Op } \sigma \ pxl \ Tl$, 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:

$$\begin{aligned} \text{datatype (sort, opsym, rlsym, psort, param) atm} = \\ \text{Pcond (param list} \rightarrow \text{bool) (psort list) (pvar list) } \mid \\ \text{Eq ((sort, opsym) trm) ((sort, opsym) trm) } \mid \\ \text{Rl rlsym (pvar list) (((sort, opsym) trm) list)} \end{aligned}$$

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 $\text{Pcond } R \ psl \ pxl$. 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 $\text{Eq } s \ T_1 \ T_2$. 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 $\text{Rl } \pi \ pxl \ Tl$. 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:

$$\begin{aligned} \text{datatype (sort, opsym, rlsym, psort, param) hcl} = \\ \text{Horn (((sort, opsym, rlsym, psort, param) atm) list) } \\ \text{((sort, opsym, rlsym, psort, param) atm)} \end{aligned}$$

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 $(\text{sort, opsym}) \text{ trm}$.

4.2 Signatures

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

$$\begin{aligned} \text{locale Signature} = \\ \text{fixes stOf : opsym} \rightarrow \text{sort} \\ \text{and arOf : opsym} \rightarrow \text{sort list} \quad \text{and arOfP : opsym} \rightarrow \text{psort list} \\ \text{and rarOf : rlsym} \rightarrow \text{sort list} \quad \text{and rarOfP : rlsym} \rightarrow \text{psort list} \\ \text{and params : psort} \rightarrow \text{param} \rightarrow \text{bool} \\ \text{and prels : ((param list} \rightarrow \text{bool)} \times \text{psort list) set} \end{aligned}$$

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 $(R, psl) \in \text{prels}$, 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 `trms : sort → trm → bool`, by requiring that operation symbols are applied according to their arities.

A model is a tuple $(\alpha, \text{intSt}, \text{intOp}, \text{intRl})$, where:

- α is the carrier type,
- `intSt : sort → α → bool` classifies the elements of α according to sorts;
- `intOp : opsym → param list → α list → α` interprets the operation symbols as parameterized operations on α ;
- `intRl : rlsym → param list → α list → 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:

$$\forall \sigma \text{ pl } al. \text{list_all2 params (arOfP } \sigma) \text{ pl} \wedge \text{list_all2 intSt (arOf } \sigma) \text{ al} \rightarrow \text{intSt (stOf } \sigma) (\text{intOp } \sigma \text{ pl } al).$$

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

$$\text{intTrm intOp intPvar intVar (Var } s \ x) = \text{intVar } s \ x$$

$$\text{intTrm intOp intPvar intVar (Op } \sigma \ \text{pxl } \text{TL}) =$$

$$\text{intOp } \sigma \ (\text{map2 intPvar (arOfP } \sigma) \ \text{pxl}) \ (\text{map (intTrm intOp intPvar intVar) } \text{TL})$$

$$\text{satAtm intOp intEq intRl intPvar intVar (Pcond } R \ \text{psl } \ \text{pxl}) \leftrightarrow R \ (\text{map2 intPvar psl } \ \text{pxl})$$

$$\text{satAtm intOp intEq intRl intPvar intVar (Eq } s \ T_1 \ T_2) \leftrightarrow$$

$$\text{intEq (intTrm intOp intPvar intVar } T_1) \ (\text{intTrm intOp intPvar intVar } T_2)$$

$$\text{satAtm intOp intEq intRl intPvar intVar (Rl } \pi \ \text{pxl } \ \text{TL}) \leftrightarrow$$

$$\text{intRl } \pi \ (\text{map2 intPvar (rarOfP } \pi) \ \text{pxl}) \ (\text{map (intTrm intOp intPvar intVar) } \ \text{TL})$$

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 : α → 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: $\text{compat } \text{intSt } \text{intOp} \wedge (\forall ps \text{ px. params } ps (\text{intPvar } ps \text{ px})) \wedge (\forall s x. \text{intSt } s (\text{intVar } s \text{ x})) \rightarrow (\text{trms } s \text{ T} \rightarrow \text{intSt } s (\text{intTrm } \text{intOp } \text{intPvar } \text{intVar } \text{T}))$.

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 quotient terms w.r.t. 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:

$$\begin{aligned} \text{satHcl } \text{intSt } \text{intOp } \text{intEq } \text{intRl } (\text{Horn } \text{atml } \text{atm}) \leftrightarrow \\ \forall \text{intPvar } \text{intVar. } (\forall ps \text{ px. params } ps (\text{intPvar } ps \text{ px})) \wedge (\forall s x. \text{intSt } s (\text{intVar } s \text{ x})) \wedge \\ \text{list_all } (\text{satAtm } \text{intOp } \text{intEq } \text{intRl } \text{intPvar } \text{intVar}) \text{atml} \rightarrow \\ \text{satAtm } \text{intOp } \text{intEq } \text{intRl } \text{intPvar } \text{intVar } \text{atm} \end{aligned}$$

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 quotienting ground terms w.r.t. an equivalence relation. Hence its carrier will be the following type of ‘‘Horn terms’’ defined to be sets of ground terms:

$$\text{type_synonym } (\text{opsym, param}) \text{ htrm} = ((\text{opsym, param}) \text{ gtrm}) \text{ set}$$

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:

$$\begin{aligned} \text{locale HornTheory} = \text{Signature} + \text{fixes HCL} : \text{hcl set} \\ \text{assumes } \forall \text{hcl} \in \text{HCL. wf hcl} \text{ and } \forall s. \text{reach } s \text{ and } \forall ps. \exists p. \text{params } ps \text{ p} \end{aligned}$$

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 $\text{Eq } s \text{ T}_1 \text{ T}_2$, 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

$\text{htrms} : \text{sort} \rightarrow \text{htrm} \rightarrow \text{bool}$, $\text{Hop} : \text{opsym} \rightarrow \text{param list} \rightarrow \text{htrm list} \rightarrow \text{htrm}$ and $\text{Hrel} : \text{rlsym} \rightarrow \text{param list} \rightarrow \text{htrm list} \rightarrow \text{bool}$.

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

lemma: $\text{hcl} \in \text{HCL} \rightarrow \text{satHcl gtrms Gop Geq Grel hcl}$

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

theorem satisfaction: $\text{hcl} \in \text{HCL} \rightarrow \text{satHcl htrms Hop (=) Hrel hcl}$

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

theorem induction: $(\forall \sigma \text{ pl } Hl. \text{list_all2 params (arOfP } \sigma) \text{ pl} \wedge \text{list_all2 htrms (arOf } \sigma) Hl \wedge \text{list_all2 } \varphi \text{ (arOf } \sigma) Hl \rightarrow \varphi \text{ (stOf } \sigma) \text{ (Hop } \sigma \text{ pl } Hl)) \rightarrow (\text{htrms } s \text{ H} \rightarrow \varphi \text{ s } H)$.

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 $\text{giter} : (\text{opsym} \rightarrow \text{param list} \rightarrow \alpha \text{ list} \rightarrow \alpha) \rightarrow \text{gtrm} \rightarrow \alpha$ that interprets ground terms with an operation symbol interpretation on a type α , as $\text{giter intOp (Gop } \sigma \text{ pl } Tl) = \text{intOp } \sigma \text{ pl (map (giter intOp) Tl)}$. Then we lift giter to htrm equivalence classes, giving $\text{iter} : (\text{opsym} \rightarrow \text{param list} \rightarrow \alpha \text{ list} \rightarrow \alpha) \rightarrow \text{htrm} \rightarrow \alpha$. If $(\alpha, \text{intSt}, \text{intOp}, \text{intRl})$ 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: $\text{compat intSt intOp} \wedge (\forall \text{ hcl} \in \text{HCL}. \text{satHcl intSt intOp (=) intRl hcl}) \rightarrow \text{htrms } s \text{ H} \rightarrow \text{intSt } s \text{ (iter intOp } H)$

theorem iteration: $\text{compat intSt intOp} \wedge (\forall \text{ hcl} \in \text{HCL}. \text{satHcl intSt intOp (=) intRl hcl}) \rightarrow \text{iter intOp (Hop } \sigma \text{ pl } Hl) = \text{intOp } \sigma \text{ pl (map (iter intOp) Hl)}$

theorem it_pres: $\text{compat intSt intOp} \wedge (\forall \text{ hcl} \in \text{HCL}. \text{satHcl intSt intOp (=) intRl hcl}) \rightarrow \text{Hrel } \pi \text{ pl } Hl \rightarrow \text{intRl } \pi \text{ pl (map (iter intOp) Hl)}$.

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 still 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 [20] for a description 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 $ps\ p \leftrightarrow (ps = a \wedge \text{isInl } p) \vee (ps = b \wedge \text{isInr } p)$;
- prels = $\{(\text{dif}_2, [a, a])\}$, where dif_2 is the function sending any list of two parameters of the form $[\text{Inl } a_1, \text{Inl } a_2]$ 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, $x \neq y \wedge \text{fresh } x\ s \longrightarrow \text{Lam } y\ s = \text{Lam } x\ (s\ [\text{Var } x / y])$ becomes Horn [atm₁, atm₂] atm₃, where:

- we take x and y to be distinct elements of pvar and s to be some element of var;
- atm₁ = Pcond $\text{dif}_2\ [a, a]\ [x, y]$,
- atm₂ = Rl fr $[x]\ [\text{Var } \text{lt } s]$ and atm₃ = Eq lt $T_1\ T_2$, with $T_1 = \text{Op } \text{lam } [y]\ [\text{Var } \text{lt } s]$ and $T_2 = \text{Op } \text{lam } [x]\ [\text{Op } \text{subst } [y]\ [\text{Var } \text{lt } s, \text{Op } \text{var } [x]\ []]]$.

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 App $s_1\ s_2$ instead of Hop app [] $[s_1, s_2]$ for λ -term application, and $\forall x\ y\ s. x \neq y \wedge \text{fresh } x\ s \longrightarrow \text{Lam } y\ s = \text{Lam } x\ (s\ [\text{Var } x / y])$ instead of satHcl *intSt intOp* (=) *intRl* (Horn [atm₁, atm₂] atm₃).

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 htrms 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 $\varphi : A \rightarrow \text{bool}$ and $\psi : B \rightarrow \text{bool}$, $\varphi \otimes \psi : A \times B \rightarrow \text{bool}$ is defined

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

$\text{htrms lt} : \text{htrm} \rightarrow \text{bool}$	lterm
$\text{isInl} : \text{param} \rightarrow \text{bool}$	α
$\text{isInr} : \text{param} \rightarrow \text{bool}$	β
$(\text{list_all2 params (arOfP app)}) \otimes (\text{list_all2 htrms (arOf app)}) :$ $\text{param list} \times \text{htrm list} \rightarrow \text{bool}$	$\text{lterm} \times \text{lterm}$
$(\text{list_all2 params (arOfP lam)}) \otimes (\text{list_all2 htrms (arOf lam)}) :$ $\text{param list} \times \text{htrm list} \rightarrow \text{bool}$	$\alpha \times \text{lterm}$

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 $(\alpha, \beta)\text{lterm}$. 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 operations³—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 $\text{arOf app} = [\text{lt}, \text{lt}]$, we see that $\text{list_all} (\text{arOf app}) H1$ requires that $H1$ have the form $[H_1, H_2]$ such that $\text{htrms lt } H_1$ and $\text{htrms lt } H_2$ hold—thus, the lists boil down to pairs of Horn terms of sort lt , hence correspond bijectively to $\text{lterm} \times \text{lterm}$.

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., $\text{App} : \text{lterm} \times \text{lterm} \rightarrow \text{lterm}$ is defined as the image of $\text{Hop app} : \text{param list} \times \text{htrm list} \rightarrow \text{trmHCL}$ restricted according to the suitable predicates. Thus, App corresponds to Hop app under the lifted bijection to $\text{lterm} \times \text{lterm} \rightarrow \text{lterm}$ from the set of elements of $\text{param list} \times \text{htrm list} \rightarrow \text{trmHCL}$ for which the predicate $(\text{list_all2 params (arOfP app)}) \otimes (\text{list_all2 htrms (arOf app)}) \Rightarrow \text{htrms (stOf app)}$ holds. This set contains 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 l_2 instead of

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

list_all2:

$$\forall \varphi s H. (\forall \sigma pl HL. l_2 \text{ params } (\text{arOfP } \sigma) pl \wedge l_2 \text{ htrms } (\text{arOf } \sigma) HL \wedge l_2 \varphi (\text{arOf } \sigma) HL \rightarrow \varphi (\text{stOf } \sigma) (\text{Hop } \sigma pl HL)) \rightarrow \text{htrms } s H \rightarrow \varphi s H$$

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:

$$\begin{aligned} \forall \varphi s H. (\forall \varphi pl HL. l_2 \text{ params } (\text{arOfP } \text{app}) pl \wedge l_2 \text{ htrms } (\text{arOf } \text{app}) HL \wedge l_2 \varphi (\text{arOf } \text{app}) HL \rightarrow \varphi (\text{stOf } \text{app}) (\text{Hop } \text{app } pl HL)) \\ \wedge (\forall \varphi pl HL. l_2 \text{ params } (\text{arOfP } \text{lam}) pl \wedge l_2 \text{ htrms } (\text{arOf } \text{lam}) HL \wedge l_2 \varphi (\text{arOf } \text{lam}) HL \rightarrow \varphi (\text{stOf } \text{lam}) (\text{Hop } \text{lam } pl HL)) \\ \rightarrow \text{htrms } s H \rightarrow \varphi s H \end{aligned}$$

Computing the values of the sort and arity functions, this becomes:

$$\begin{aligned} \forall \varphi s H. (\forall \varphi pl HL. l_2 \text{ params } [] pl \wedge l_2 \text{ htrms } [!t, !t] HL \wedge l_2 \varphi [!t, !t] HL \rightarrow \varphi !t (\text{Hop } \text{app } pl HL)) \\ \wedge (\forall \varphi pl HL. l_2 \text{ params } [a] pl \wedge l_2 \text{ htrms } [!t] HL \wedge l_2 \varphi [!t] HL \rightarrow \varphi !t (\text{Hop } \text{lam } pl HL)) \\ \rightarrow \text{htrms } s H \rightarrow \varphi s H \end{aligned}$$

By isomorphic transfer over the aforementioned extended bijections, we obtain:

$$(\forall H_1 H_2. \varphi H_1 \wedge \varphi H_2 \rightarrow \varphi (\text{App } H_1 H_2)) \wedge (\forall x H. \varphi H \rightarrow \varphi (\text{Lam } x H)) \rightarrow \varphi H.$$

In this step the $l_2 \text{ params}$, $l_2 \text{ htrms}$ 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. For details we refer to the first author's M.Sc. thesis [20]. 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 that correspond to the conclusion.

Isomorphic transfer is implemented [20] in the form of an algorithmic rule system. 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 $f (\text{Cons } t \text{ } ts)$ 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 $\psi_3^{-1} ((\psi_1 f) (\psi_2 t)) \text{ } ts$. 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 [24] 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 an 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 §3).

In the homotopy interpretation of type theory there is a recent trend [22] 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, Armin Heller, and the reviewers for providing many comments that helped improve the presentation, 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. The work reported here is supported by the DFG project Ni 491/13–2 (part of the DFG priority program Reliably Secure Software Systems–RS3).

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. Kaliszky 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, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002.
18. M. Norrish. Recursive function definition for types with binders. In *TPHOLs*, pp. 241–256, 2004.
19. A. Popescu and E. L. Gunter. Recursion principles for syntax with bindings and substitution. In *ICFP*, pp. 346–358, 2011.
20. 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>.
21. A. Schropp and A. Popescu. Nonfree datatypes: metatheory, implementation and examples. http://bitbucket.org/isaspecops/nonfree-data/downloads/cpp2013_bundle.zip.
22. 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/>.
23. V. Tannen and R. Subrahmanyam. Logical and computational aspects of programming with sets/bags/lists. In *ICALP*, pp. 60–75, 1991.
24. 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.