Structured Formal Development with Quotient Types in Isabelle/HOL

Maksym Bortin[1] and Christoph Lüth[2]

[1] Universität Bremen, Department of Mathematics and Computer Science
maxim@informatik.uni-bremen.de
[2] Deutsches Forschungszentrum für Künstliche Intelligenz, Bremen
christoph.lueth@dfki.de

Abstract. General purpose theorem provers provide sophisticated proof methods, but lack some of the advanced structuring mechanisms found in specification languages. This paper builds on previous work extending the theorem prover Isabelle with such mechanisms. A way to build the quotient type over a given base type and an equivalence relation on it, and a generalised notion of folding over quotiented types is given as a formalised high-level step called a design tactic. The core of this paper are four axiomatic theories capturing the design tactic. The applicability is demonstrated by derivations of implementations for finite multisets and finite sets from lists in Isabelle.

1 Introduction

Formal development of correct systems requires considerable design and proof effort in order to establish that an implementation meets the required specification. General purpose theorem provers provide powerful proof methods, but often lack the advanced structuring and design concepts found in specification languages, such as *design tactics* [20]. A design tactic represents formalised development knowledge. It is an abstract development pattern proven correct once and for all, saving proof effort when applying it and guiding the development process. If theorem provers can be extended with similar concepts without loss of consistency, the development process can be structured within the prover. This paper is a step in this direction. Building on previous work [2] where an approach to the extension of the theorem prover Isabelle [15] with theory morphisms has been described, the contributions of this paper are the representation of the well-known type quotienting construction and its extension with a generalised notion of folding over the quotiented type as a design tactic. Two applications of the tactic are presented, demonstrating the viability of our approach.

The paper is structured as follows: we first give a brief overview of Isabelle and theory morphisms to keep the paper self-contained. Sect. 3 describes the four theories which form the design tactic, giving more motivation for it and sketching the theoretical background. Further, Sect. 4 shows how the design tactic can be applied in order to derive implementations of finite multisets and finite sets. Finally, Sect. 5 contains conclusions and sketches future work.

S. Autexier et al. (Eds.): AISC/Calculemus/MKM 2010, LNAI 6167, pp. 34–48, 2010.

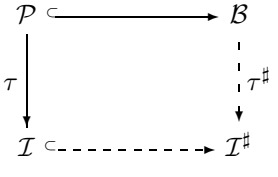
2 Isabelle and Theory Morphisms

Isabelle is a logical framework and LCF-style theorem prover, where the meta-level inference system implements an intuitionistic fragment of the higher order logic extended with Hindley-Milner polymorphism and type classes.

Isabelle, and other LCF provers, structure developments in hierarchical theories. This goes well with the predominant development paradigm of conservative extension, which assures consistency when developing large theories from a small set of axioms (such as HOL or ZF). A different approach, going back to Burstall and Goguen [3], is to use *theory morphisms* as a structuring device. Instead of one large theory we have lots of *little theories* [8], related by theory morphisms. Structuring operations are given by *colimits* of diagrams of morphisms [9], of which (disjoint and non-disjoint) unions and parametrisation are special cases. Early systems in this spirit include IMPS [8] and Clear [4], later systems the OBJ family with its most recent offspring CafeOBJ [6], and the SpecWare system [21]. An extension of the early Edinburgh LCF with theory morphisms was described in [18], but never integrated into later LCF systems. A recent development in this vein is a calculus for reasoning in such structured developments [13], as used in the CASL specification languages [14].

The morphism extension package for Isabelle [2] provides implementations of key concepts such as signature and theory morphisms, and seamlessly extends Isabelle's top-level language Isar with the commands necessary to express these notions; we will use these commands in the following. A crucial property is that any theory morphism $\tau : \mathcal{T} \longrightarrow \mathcal{T}'$ from a theory $\mathcal{T}$ to a theory $\mathcal{T}'$ firstly induces the homomorphic extension $\overline{\sigma}_\tau$ of the underlying signature morphism σ_τ to propositions, and secondly the extension $\overline{\tau}$ of τ to proof terms. This allows the translation of any theorem ϕ in $\mathcal{T}$ to a theorem $\overline{\sigma}_\tau(\phi)$ in $\mathcal{T}'$, translating the proof π of ϕ to $\overline{\tau}(\pi)$ and replaying it in $\mathcal{T}'$. It is syntactically represented in Isar by the command **translate-thm** ϕ **along** τ.[1]

Furthermore, the approach gives a simple notion of a parameterised theory, extending the theory hierarchy: a theory $\mathcal{B}$ is parameterised by $\mathcal{P}$ (denoted $\langle \mathcal{P}, \mathcal{B} \rangle$) if an inclusion morphism $\iota : \mathcal{P} \hookrightarrow \mathcal{B}$ exists or, in other words, $\mathcal{B}$ imports $\mathcal{P}$; an instantiation of $\langle \mathcal{P}, \mathcal{B} \rangle$ is given by a theory morphism $\tau : \mathcal{P} \longrightarrow \mathcal{I}$ as shown by the following diagram



(1)

[1] The current release 0.9.1 for Isabelle2009-1 can be downloaded at `http://www.informatik.uni-bremen.de/~cxl/awe` and all theories presented here can be found in the directory `Examples/Quotients`.

where the extended theory $\mathcal{I}^\sharp$ and the dashed morphisms are automatically derived. In other words, the resulting theory $\mathcal{I}^\sharp$ is the pushout of the diagram, and is computed via the Isar command **instantiate-theory** $\mathcal{B}$ **by-thymorph** τ.

3 Folding Quotient Types Using Hylomorphisms

A design tactic can be encoded as a parametrisation $\langle \mathcal{P}, \mathcal{B} \rangle$, where $\mathcal{P}$ contains formal parameters and their axiomatic specifications, and $\mathcal{B}$ contains deductions in form of definitions and theorems using the formal parameters and axioms imported from $\mathcal{P}$. In this section, we introduce a design tactic which performs two constructions: firstly, it constructs the quotient of a type with respect to an equivalence relation, and secondly, it gives a generic mechanism to define 'folding' functions on the quotient type. The tactic has two parameters: the type with the equivalence relation, and the parameters of the fold. Thus, the design tactic comprises two parametrisations in the sense of (1) above:

$$\mathit{QuotientType\text{-}Param} \hookrightarrow \mathit{QuotientType} \hookrightarrow \mathit{Fold\text{-}Param} \hookrightarrow \mathit{Fold} \tag{2}$$

The first parametrisation $\langle \mathit{QuotientType\text{-}Param}, \mathit{QuotientType} \rangle$ comprises the basic machinery regarding equivalence classes, class operations, quotient types and congruences. The core of the design tactic is the second parametrisation $\langle \mathit{Fold\text{-}Param}, \mathit{Fold} \rangle$, describing how to construct hylomorphisms on quotient types, and will be explicitly described in Sect. 3.5 and Sect. 3.6.

3.1 Quotient Types

Roughly, any equivalence relation $\simeq$ on a type τ induces a partition on $\mathit{Univ}(\tau)$, i.e. on the set containing all elements of this type. Elements of this partition are predicates and correspond to the $\simeq$-equivalence classes. This is a well-known technique. Indeed, the quotient type is a powerful construction, and implemented in many theorem provers, either axiomatically [16] (for NuPRL) or as a derived construction. The former always bears the danger of inconsistencies (see [10] for a model construction; [5] presents an implementation for Coq); the latter is made easier by the presence of a choice operator and extensionality, allowing quotient types in HOL [12] or Isabelle [19,17]. However, the main novelty here is the way in which a fold operator is defined on the quotient types as a hylomorphism in the abstract setting of parameterised theories, combining the advantages of the little-theories approach with a construction motivated from type theory.

3.2 The Theory *QuotientType-Param*

This theory declares an unary type constructor T and a relation $\simeq$ as a polymorphic constant, together with axioms specifying $\simeq$ as an equivalence relation:

typedecl $\alpha\ \mathsf{T}$
const $\_ \simeq \_ :: (\alpha\ \mathsf{T} \times \alpha\ \mathsf{T})\ \mathsf{set}$
axioms (E1) : $s \simeq s$
(E2) : $s \simeq t \Longrightarrow t \simeq s$
(E3) : $s \simeq t \Longrightarrow t \simeq u \Longrightarrow s \simeq u$

3.3 The Theory *QuotientType*

We are interested in the partition of $Univ(\alpha\mathsf{T})$: $Q_\simeq \equiv \{\{v|u \simeq v\}|u \in Univ(\alpha\mathsf{T})\}$, and introduce the new unary quotient type constructor $\mathsf{T}/_\simeq$

typedef $\alpha\ \mathsf{T}/_\simeq = Q_\simeq$

Further, we define the class operations $\textit{class-of}_\simeq :: \alpha\ \mathsf{T} \Rightarrow \alpha\ \mathsf{T}/_\simeq$ (as usually denoted by $[\_]_\simeq$) and $\textit{repr}_\simeq :: \alpha\ \mathsf{T}/_\simeq \Rightarrow \alpha\ \mathsf{T}$, such that the following familiar properties, essential for equivalence relations and quotients, can be proven:

$$([s]_\simeq = [t]_\simeq) = (s \simeq t) \tag{3}$$

$$[\textit{repr}_\simeq\ (q)]_\simeq = q \tag{4}$$

$$\textit{repr}_\simeq\ ([s]_\simeq) \simeq s \tag{5}$$

The crucial observation is that the entire development from now on relies only on these three basic properties of the class operations, i.e. we essentially abstract over the particular representation of quotients.

A function $f :: \alpha\ \mathsf{T} \Rightarrow \beta$ is called a $\simeq$-*congruence* if it respects $\simeq$ [17]; this is expressed by the predicate $\textit{congruence}_\simeq :: (\alpha\ \mathsf{T} \Rightarrow \beta)\ \mathsf{set}$ defined as

$$\textit{congruence}_\simeq \equiv \{f \mid \forall\, s\ t.\ \neg\ s \simeq t\ \vee\ f\ s = f\ t\}$$

Moreover, the higher order function $\_^{\mathsf{T}/\simeq} :: (\alpha\ \mathsf{T} \Rightarrow \beta) \Rightarrow (\alpha\ \mathsf{T}/_\simeq \Rightarrow \beta)$, which factors any $\simeq$-congruence $f :: \alpha\ \mathsf{T} \Rightarrow \beta$ through the projection $\textit{class-of}_\simeq$, i.e. such that

$$f \in \textit{congruence}_\simeq \Longrightarrow f^{\mathsf{T}/\simeq}\ [s]_\simeq = f\ s \tag{6}$$

holds, is defined as $f^{\mathsf{T}/\simeq} \equiv f \circ \textit{repr}_\simeq$. The direction $\Longleftarrow$ in (6) can be then shown as well, emphasising that the congruence condition is also necessary. Further, let $g :: \alpha\ \mathsf{T} \Rightarrow \alpha\ \mathsf{T}$ be a function. The instantiation of f by $\textit{class-of}_\simeq \circ g$ in (6) gives

$$(\textit{class-of}_\simeq \circ g) \in \textit{congruence}_\simeq \Longrightarrow (\textit{class-of}_\simeq \circ g)^{\mathsf{T}/\simeq}\ [s]_\simeq = [g\ s]_\simeq \tag{7}$$

All these derived properties are well-known, but note that the complete development here is parameterised over the type constructor T and the relation $\simeq$, and thus can be re-used in a variety of situations.

3.4 Defining Functions over Quotient Types

In order to define a function f on the quotient type $\alpha\ \mathsf{T}/_\simeq$, we have to show that f agrees with the equivalence relation $\simeq$. Equation (6) gives us sufficient conditions for this. The following theory development makes use of this for a design tactic which axiomatises sufficient conditions to conveniently define linear recursive functions, or *hylomorphisms* [7], on the quotient type. We first motivate the development by sketching the special case of lists, and then generalise to arbitrary data types.

In Isabelle, the parameterised type $\alpha\,\mathsf{list}$ of lists of elements of type α is freely generated by the constructors $Nil :: \alpha\,\mathsf{list}$ and the infix operator $\# :: \alpha \Rightarrow \alpha\,\mathsf{list} \Rightarrow \alpha\,\mathsf{list}$. Suppose we would like to prove

$$(\forall ys)\frac{xs \sim ys \qquad (f,e) \in C}{foldr\ f\ e\ xs = foldr\ f\ e\ ys}$$

by structural induction on the list xs, where $\sim$ is an equivalence relation on lists, and f and e are additionally restricted by some (usually non-trivial) side condition C. The crucial point would be the induction step, where based on the assumption $x\#xs \sim ys$ we need to find some list zs satisfying $xs \sim zs$ and, moreover allowing us to conclude $f\ x\ (foldr\ f\ e\ zs) = foldr\ f\ e\ ys$. In many cases such zs can be computed by a function $Transform\ x\ xs\ ys$ constructing a list which satisfies the desired properties under the premises $x\#xs \sim ys$ and $(f,e) \in C$; thus, we can say the proof is parameterised over the function *Transform*.

Hylomorphisms are particular kinds of recursive functions which can be expressed in terms of (co-)algebras for the same type. Consider a parameterised type $\alpha\ \Sigma$, together with an action Σ on functions (normally called *map*; the map on types and functions together form a functor). Then an *algebra* for Σ is a type γ and a function $A :: \gamma\,\Sigma \Rightarrow \gamma$, a *coalgebra* is a type β and a function $B :: \beta \Rightarrow \beta\,\Sigma$, and the solution of the *hylo-equation* [7]

$$\phi = A \circ \Sigma\phi \circ B \tag{8}$$

is a function $\phi :: \beta \Rightarrow \gamma$, called the *hylomorphism* from B to A. Hylomorphisms correspond to linear recursive functions and can be compiled efficiently; hence, deriving them via a general design tactic is relevant.

In the case of lists, the list signature is represented by the type $(\alpha,\beta)\,\Sigma_{\mathsf{list}} \stackrel{def}{=} \mathbf{1} + \alpha \times \beta$ (as usual, $\times$ denotes the product and $+$ the disjoint sum of two types), together with the map function $\Sigma_{\mathsf{list}} :: (\beta \Rightarrow \gamma) \Rightarrow (\alpha,\beta)\,\Sigma_{\mathsf{list}} \Rightarrow (\alpha,\gamma)\,\Sigma_{\mathsf{list}}$ defined by the equations

$$\begin{aligned}\Sigma_{\mathsf{list}}\ f\ (\iota_L\ *) &= \iota_L\ *\\ \Sigma_{\mathsf{list}}\ f\ (\iota_R\ (u,x)) &= \iota_R\ (u, f\ x)\end{aligned}$$

The type $\alpha\ \mathsf{list}$ from above, together with the function $in_{\mathsf{list}} :: (\alpha, \alpha\,\mathsf{list})\,\Sigma_{\mathsf{list}} \Rightarrow \alpha\,\mathsf{list}$, defined in the obvious way sending $\iota_L\ *$ to Nil and $\iota_R\ (u,x)$ to $u\#x$, forms the initial Σ_{list}-algebra. Its inverse is the function out_{list}, which forms a Σ_{list}-coalgebra, i.e. we have the right-inverse property: $in_{\mathsf{list}} \circ out_{\mathsf{list}} = id_{\mathsf{list}}$. The initiality of in_{list} means that any Σ_{list}-algebra $A :: (\alpha,\beta)\,\Sigma_{\mathsf{list}} \Rightarrow \beta$ determines the unique algebra homomorphism $\phi_A : \alpha\ \mathsf{list} \Rightarrow \beta$, i.e.

$$\phi_A \circ in_{\mathsf{list}} = A \circ \Sigma_{\mathsf{list}}\phi_A \tag{9}$$

holds. If we compose both sides of (9) with out_{list} on the right and use the right-inverse property of out_{list}, we obtain the fact that ϕ_A satisfies the hylo-equation (8), i.e. is the hylomorphism from out_{list} to A.

The unique function ϕ_A can be defined using *foldr*. That *foldr* determines hylomorphisms from out_{list} to any Σ_{list}-algebra is an important observation, because in the following we want to explore the congruence properties of hylomorphisms. Taking also into account that many structures can be implemented via quotients over lists, we obtain the possibility to extend *foldr* to $foldr^{\mathsf{list}/\sim}$ and to calculate with $foldr^{\mathsf{list}/\sim}$ based on the numerous properties of *foldr*.

3.5 The Theory *Fold-Param*

We will now generalise the previous development to an arbitrary type constructor Σ, and formalise it as a parameterised theory. The parameter theory *Fold-Param* is constructed in four steps; the body theory *Fold* follows in Sect. 3.6.

(1) Representing signatures. First of all, a signature is represented by a declaration of a binary type constructor Σ, together with the two polymorphic constants representing the action of Σ on relations and mappings, respectively.

typedecl $(\alpha, \beta)\,\Sigma$
consts $\Sigma^{Rel} :: (\beta \times \gamma)\,\mathsf{set} \Rightarrow ((\alpha, \beta)\,\Sigma \times (\alpha, \gamma)\,\Sigma)\,\mathsf{set}$
$\Sigma^{Map} :: (\beta \Rightarrow \gamma) \Rightarrow (\alpha, \beta)\,\Sigma \Rightarrow (\alpha, \gamma)\,\Sigma$

Using this, the action $\Sigma^{Pred} :: \beta\,\mathsf{set} \Rightarrow ((\alpha, \beta)\,\Sigma)\,\mathsf{set}$ of Σ on predicates over β can be defined by $\Sigma^{Pred} \equiv mono^P \circ \Sigma^{Rel} \circ mono^E$, where $mono^E :: \alpha\,\mathsf{set} \Rightarrow (\alpha \times \alpha)\,\mathsf{set}$ is the embedding of predicates into relations in form of monotypes, and $mono^P :: (\alpha \times \alpha)\,\mathsf{set} \Rightarrow \alpha\,\mathsf{set}$ the corresponding projection. Furthermore, using Σ^{Rel} we define the extension $\simeq_\Sigma$ of our formal parameter $\simeq$ from *QuotientType-Param* simply by $\simeq_\Sigma \equiv \Sigma^{Rel} \simeq$.

Finally, the rule connecting the actions of Σ is given by axiom (F1), where $R \backslash_. S$ is defined to be $\{x \,|\, \forall a.\ (x, a) \notin R \vee (x, a) \in S\}$, i.e. it is a sort of factoring of the relation R through the relation S:

axiom (F1) : $\Sigma^{Pred}(\simeq \backslash_.\ ker\, f) \quad \subseteq \quad \simeq_\Sigma \backslash_.\ ker\,(\Sigma^{Map}\ f)$

(2) The parameter coalgebra. Next, we specify the constant c_T representing a Σ-coalgebra with the domain $\alpha\,\mathsf{T}$ satisfying property (F2), where $P :: (\alpha\,\mathsf{T})\,\mathsf{set}$ is an arbitrary predicate and $f^{-1}\langle S\rangle$ denotes the preimage of a function f under a predicate S, i.e. $\{x \mid f\ x \in S\}$:

const $c_\mathsf{T} :: \alpha\,\mathsf{T} \Rightarrow (\alpha, \alpha\,\mathsf{T})\,\Sigma$
axiom (F2) : $c_\mathsf{T}^{-1}\langle \Sigma^{Pred}\ P\rangle \subseteq P \quad \Longrightarrow \quad Univ(\alpha\,\mathsf{T}) \subseteq P$

The axiom (F2) is a slightly adapted characterisation of so-called Σ-*reductive* coalgebras, which can be found in [7]. It essentially ensures that the sequence $s_0, s_1 \circ s_0, s_2 \circ s_1 \circ s_0, \ldots$ with $s_0 = c_\mathsf{T}$ and $s_{n+1} = \Sigma^{Map}\ s_n$, is not infinite and reaches some fixed point $s_k \circ \ldots \circ s_0$ with $k \in \mathbb{N}$. Thus, it also captures an induction principle.

(3) The hylomorphism parameter. The higher-order constant *Fold* is required to return a hylomorphism *Fold A* from c_T to A for any $A \in$ *FoldCond*:

const $\mathit{Fold} :: ((\alpha, \beta)\, \Sigma \Rightarrow \beta) \Rightarrow \alpha\, \mathsf{T} \Rightarrow \beta$
axiom (F3) : $A \in \mathit{FoldCond} \implies \mathit{Fold}\, A = A \circ\ \Sigma^{\mathit{Map}}(\mathit{Fold}\, A)\ \circ c_{\mathsf{T}}$

The predicate *FoldCond* on Σ-algebras is completely unspecified at this point, and therefore can be arbitrarily instantiated whenever the tactic is applied.

(4) Transformation function. Finally, we require a transformation function satisfying the properties (F4) and (F5), where *TransformCond* is another Σ-algebra predicate for which merely (F6) is required:

const $\mathit{Transform} :: (\alpha, \alpha\, \mathsf{T})\, \Sigma \Rightarrow (\alpha, \alpha\, \mathsf{T})\, \Sigma \Rightarrow (\alpha, \alpha\, \mathsf{T})\, \Sigma$
axioms (F4) : $s \simeq t \implies c_{\mathsf{T}}\, s \simeq_{\Sigma} \mathit{Transform}\, (c_{\mathsf{T}}\, s)\, (c_{\mathsf{T}}\, t)$
(F5) : $A \in \mathit{TransformCond} \implies s \simeq t \implies$
$A\, (\Sigma^{\mathit{Map}}\, (\mathit{Fold}\, A)\, (\mathit{Transform}\, (c_{\mathsf{T}}\, s)\, (c_{\mathsf{T}}\, t))) = \mathit{Fold}\, A\, t$
(F6) : $\mathit{TransformCond} \subseteq \mathit{FoldCond}$

Transform can be considered as a function transforming its second argument w.r.t. its first argument. The axiom (F5) essentially requires that if both arguments comprise images of two elements, which are in the $\simeq$ relation, then *Transform* respects the kernel of $A \circ \Sigma^{\mathit{Map}}\, (\mathit{Fold}\, A)$.

3.6 The Theory *Fold*

The operations and conditions, specified in *Fold-Param* are sufficient in order to derive the congruence property for *Fold A* for any Σ-algebra A, satisfying the transformation condition *TransformCond*. To this end, the theory *Fold* proves the following central property:

Theorem 1. $A \in \mathit{TransformCond} \implies \mathit{Fold}\, A \in \mathit{congruence}_{\simeq}$

Proof. The condition $\mathit{Fold}\, A \in \mathit{congruence}_{\simeq}$ can be equivalently restated using the factoring operator by $\mathit{Univ}(\alpha\mathsf{T}) \subseteq \simeq \setminus\!. \mathit{ker}(\mathit{Fold}\, A)$, such that we can proceed by induction using the reductivity axiom (F2). Further, by monotonicity of the preimage operator and the axiom (F1) we have then to show

$$c_{\mathsf{T}}^{-1} \langle \simeq_{\Sigma} \setminus\!. \mathit{ker}\, (\Sigma^{\mathit{Map}}\, (\mathit{Fold}\, A)) \rangle \quad \subseteq \quad \simeq \setminus\!. \mathit{ker}\, (\mathit{Fold}\, A)$$

Unfolding the definitions of the factoring and preimage operators, this yields the ultimate goal: $\mathit{Fold}\, A\, s = \mathit{Fold}\, A\, t$ for any s, t of type $\alpha\, \mathsf{T}$, such that $s \simeq t$ and

$$(\forall\, u)\ \frac{c_{\mathsf{T}}\, s\ \simeq_{\Sigma}\ u}{\Sigma^{\mathit{Map}}\, (\mathit{Fold}\, A)\, (c_{\mathsf{T}}\, s)\ =\ \Sigma^{\mathit{Map}}\, (\mathit{Fold}\, A)\, u} \tag{10}$$

hold. This can be shown as follows

$$\begin{aligned} \mathit{Fold}\, A\, s &= A\, (\Sigma^{\mathit{Map}} (\mathit{Fold}\, A)\, (c_{\mathsf{T}}\, s)) \\ &= A\, (\Sigma^{\mathit{Map}}\, (\mathit{Fold}\, A)\, (\mathit{Transform}\, (c_{\mathsf{T}}\, s)\, (c_{\mathsf{T}}\, t))) \\ &= \mathit{Fold}\, A\, t \end{aligned}$$

where the first step follows by axiom (F3), the second by instantiating u in (10) with $\mathit{Transform}\, (c_{\mathsf{T}}\, s)\, (c_{\mathsf{T}}\, t)$ provided by axiom (F4), and the third by axioms (F5), (F6) and the premise $s \simeq t$. □

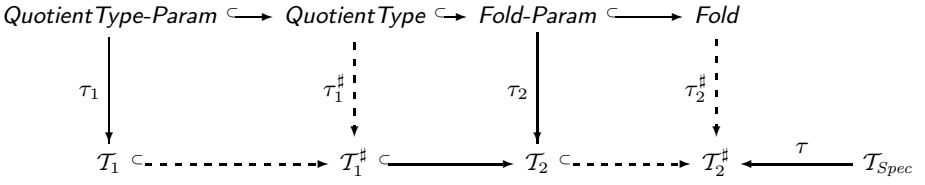


Fig. 1. Applying the fold quotient design tactic

As the immediate consequence for the function $Fold^{\mathsf{T}/\simeq} :: ((\alpha, \beta)\, \Sigma \Rightarrow \beta) \Rightarrow \alpha\, \mathsf{T}/\simeq \Rightarrow \beta$, we can finally derive from (6) via Theorem 1:

$$\frac{A \in TransformCond}{Fold^{\mathsf{T}/\simeq}\ A\ [s]_\simeq = Fold\, A\, s} \tag{11}$$

Taking for instance *foldr* for *Fold* and a list algebra A, interpreting # by a function f satisfying *TransformCond*, this means that $foldr^{\mathsf{list}/\simeq}\ A\ [x\#xs]_\simeq$ can always be replaced by $foldr\ A\ (x\#xs) = f\ x\ (foldr\ A\ xs)$, and thus by $f\ x\ (foldr^{\mathsf{list}/\simeq}\ A\ [xs]_\simeq)$.

4 Applying the Design Tactic

In this section, the presented design tactic for quotients and hylomorphism extension will be applied in order to derive implementations of bags and finite sets from lists. Recall the structure of the design tactic from (2); to apply it to a given type, we proceed in the following steps (see Fig. 1):

(i) we first provide a theory $\mathcal{T}_1$ and a morphism $\tau_1 : \mathit{QuotientType\text{-}Param} \longrightarrow \mathcal{T}_1$ which instantiates the type constructor and equivalence relation;
(ii) by instantiating *QuotientType*, we obtain $\mathcal{T}_1^\sharp$ with the quotient type;
(iii) we now extend $\mathcal{T}_1^\sharp$ into a theory $\mathcal{T}_2$, such that we can provide a theory morphism $\tau_2 : \mathit{Fold\text{-}Param} \longrightarrow \mathcal{T}_2$ instantiating the parameters for *Fold*;
(iv) by instantiating *Fold*, we obtain the theory $\mathcal{T}_2^\sharp$ with the desired function over the quotient type and the instantiated of the fold equation (11);
(v) finally, the correctness w.r.t. some axiomatic specification $\mathcal{T}_{Spec}$ is established by constructing a theory morphism $\tau : \mathcal{T}_{Spec} \longrightarrow \mathcal{T}_2^\sharp$.

Note that in Isabelle the theories $\mathcal{T}_1, \mathcal{T}_1^\sharp, \mathcal{T}_2, \mathcal{T}_2^\sharp$ are constructed as intermediate development steps of a single theory extending some base theory (in the following examples this will be the theory *List*).

4.1 Specifying Finite Sets and Bags

The rôle of theory $\mathcal{T}_{Spec}$ from step (v) above will be played by the axiomatic theories *FiniteSet-Spec* and *Bag-Spec*.

The Theory *FiniteSet-Spec*. It specifies finite sets parameterised over the type of its elements as follows. The unary type constructor finite-set is declared, together with the following polymorphic operations on it satisfying axioms (S1)– (S6):

typedecl α finite-set

consts	$\{\#\}$	:: α finite-set	- empty set
	$\_ \lessdot \_$	:: $\alpha \Rightarrow \alpha$ finite-set $\Rightarrow$ bool	- membership test
	$\_ \oplus \_$	:: $\alpha \Rightarrow \alpha$ finite-set $\Rightarrow \alpha$ finite-set	- insertion
	$\_ \ominus \_$	:: α finite-set $\Rightarrow \alpha \Rightarrow \alpha$ finite-set	- deletion
	foldSet	:: $(\alpha \Rightarrow \beta \Rightarrow \beta) \Rightarrow \beta \Rightarrow \alpha$ finite-set $\Rightarrow \beta$	- fold

axioms (S1) : $\neg\ a \lessdot \{\#\}$

(S2) : $(a \lessdot b \oplus S) = (a = b \vee a \lessdot S)$

(S3) : $(a \lessdot S \ominus b) = (a \neq b \wedge a \lessdot S)$

(S4) : $(\forall a.\ (a \lessdot S) = (a \lessdot T)) \Longrightarrow S = T$

(S5) : $foldSet\ f\ e\ \{\#\} = e$

(S6) : $f \in LeftCommuting \Longrightarrow \neg\ x \lessdot S \Longrightarrow$
$foldSet\ f\ e\ (x \oplus S) = f\ x\ (foldSet\ f\ e\ S)$

where $LeftCommuting \equiv \{f \mid \forall\ a\ b\ c.\ f\ a\ (f\ b\ c) = f\ b\ (f\ a\ c)\}$. In this specification only the last axiom ultimately eliminates arbitrary sets from the class of possible implementations of *FiniteSet-Spec*. In other words, without the last axiom the theory morphism, sending α finite-set to α set as well as $\{\#\}$ to $\emptyset$, $\lessdot$ to $\in$ and so on, is constructible.

On the other hand, *foldSet* allows us to define all the basic operations on finite sets, e.g. the cardinality of any finite set S is given by $foldSet\ (\lambda\, x\, N.\, N+1)\ 0\ S$, and the union $S \sqcup T$ by $foldSet\ (\lambda\, x\, Y.\, x \oplus Y)\ S\ T$. Moreover, we can define the translation function $toPred :: \alpha$ finite-set $\Rightarrow \alpha$ set by $foldSet\ (\lambda\, x\, P.\, \{x\} \cup P)\ \emptyset$, such that for any $S :: \alpha$ finite-set and $x :: \alpha$, $x \lessdot S$ holds iff $x \in toPred\, S$ does. Further, we can prove that the translation is also injective, and so the range of *toPred*, which is of type $(\alpha$ set$)$set, defines exactly the subset of finite predicates, isomorphic to α finite-set.

The Theory *Bag-Spec*. It specifies finite multisets in the similar manner. Here, we introduce an unary type constructor α bag together with basically the same operations on it, except that the membership function has the type $\alpha \Rightarrow \alpha$ bag $\Rightarrow$ nat and thus counts the occurrences of an element in a bag. For the insertion operation this means that we have the rules $a \lessdot a \oplus M = (a \lessdot M) + 1$ and $a \neq b \Longrightarrow a \lessdot b \oplus M = a \lessdot M$. The folding function is now consequently called *foldBag*, and has to satisfy the rule

$$f \in LeftCommuting \Longrightarrow foldBag\ f\ e\ (x \oplus M) = f\ x\ (foldBag\ f\ e\ M)$$

Similarly to finite sets, cardinality, union, intersection etc. are definable via *foldBag* in *Bag-Spec*.

4.2 Implementing Bags

The implementation of bags is on the type of α list from the Isabelle/HOL libraries. The central rôle will be played by the function $count :: \alpha \Rightarrow \alpha$ list $\Rightarrow$ nat, defined recursively as

$$count\ a\ Nil = 0$$
$$count\ a\ (b\#xs) = \begin{cases} 1 + count\ xs & \text{if } a = b \\ count\ xs & \text{otherwise} \end{cases}$$

Now, let $xs \sim ys \equiv (\forall a.\ count\ a\ xs = count\ a\ ys)$, be the equivalence relation on α list comprising the intersection of kernels of the family of functions $\langle count\ a \rangle_{a \in Univ(\alpha)}$. We can then define the following theory morphism (step (i) above)

thymorph *bag1* : *QuotientType-Param* $\longrightarrow$ *Bag*
type-map : $[\alpha\ \mathsf{T} \mapsto \alpha\ \mathsf{list}]$
op-map : $[\simeq \mapsto \sim]$

and instantiate the parameterised theory $\langle$*QuotientType-Param*, *QuotientType*$\rangle$

instantiate-theory *QuotientType* **by-thymorph** *bag1*
renames : $[\ \mathsf{T}/_{\simeq} \mapsto\ \mathsf{bag}]$

This extends the theory *Bag* (step (ii) above), introducing the new quotient type constructor $\mathsf{list}/_{\sim}$ as bag, together with the corresponding congruence predicate $congruence_{\sim}$ :: $(\alpha\ \mathsf{list} \Rightarrow \beta)\ \mathsf{set}$ and extension function $\_^{\mathsf{bag}}$, corresponding to step (ii) above. This step also gives us the theory morphism $bag1^{\sharp}$: *QuotientType* $\longrightarrow$ *Bag*, i.e. $\tau_1^{\sharp}$ in Fig. 1. Using this morphism, the corresponding instances of the properties (3) – (7) can now be translated to *Bag* along $bag1^{\sharp}$ via the **translate-thm** command. It is then routine to prove

1. $count\ x \in congruence_{\sim}$ for any x (this is in fact trivial);
2. $(class\text{-}of_{\sim} \circ (x\ \#\ \_)) \in congruence_{\sim}$ for any x;
3. $(class\text{-}of_{\sim} \circ (remove1\ x)) \in congruence_{\sim}$ for any x, where $remove1\ x\ xs$ removes the first occurrence of x from the list xs, if any;

such that the extensions of these functions from α list to α bag give us the implementations for the operations $\_ \lessdot \_$, $\_ \oplus \_$, and $\_ \ominus \_$ from *Bag-Spec*, respectively; for example the insertion $x \oplus M$ is implemented by $(class\text{-}of_{\sim} \circ (x\ \#\ \_))^{\mathsf{bag}}\ M$. It remains to give an implementation for *foldBag*.

Deriving *foldBag*. In order to proceed with step (iii), i.e. to instantiate the parameterised theory $\langle$*Fold-Param*, *Fold*$\rangle$, we need to supply actual parameters for the formal parameters in *Fold-Param*. This corresponds to construction of τ_2 in Fig. 1. First of all, the formal type parameter $(\alpha, \beta)\ \Sigma$, representing a signature, is mapped to $\mathbf{1} + \alpha \times \beta$ (the list signature). Then the parameter constants are mapped as follows:

1. the action of $\mathbf{1} + \alpha \times \beta$ on relations is defined in the standard way by
$$\Sigma^{Rel}\ R \equiv \{\iota_L *, \iota_L *\} \cup \{(\iota_R(u, x), \iota_R(u, y)) \mid (x, y) \in R, u \in Univ(\alpha)\}$$
where Σ^{Map} is exactly the same as Σ_{list}, defined in Sect. 3.5;
2. the coalgebra parameter c_{T} is instantiated by the coalgebra out_{list};

3. the hylomorphism is essentially the *foldr*-function:

$$\begin{aligned} \mathit{Fold}\ A &\equiv \mathit{foldr}\ (\lambda v\ x.\ A(\iota_R(v,x)))\ A(\iota_L *) \\ \mathit{FoldCond} &\equiv \mathit{Univ}(\mathbf{1} + \alpha \times \beta \Rightarrow \beta) \quad \text{i.e. the same as } \mathit{True} \end{aligned}$$

4. Finally, the transformation and the transformation condition are defined by

$$\mathit{Transform}\ u\ v \equiv \begin{cases} \iota_R(x, \mathit{remove1}\ x\ (y \# ys)) & \text{if } u = \iota_R(x, xs) \\ & \text{and } v = \iota_R(y, ys) \\ v & \text{otherwise} \end{cases}$$

$$\mathit{TransformCond} \equiv \{A \mid \forall\ x\ y\ z.\ \hat{A}(x, \hat{A}(y, z)) = \hat{A}(y, \hat{A}(x, z))\}$$

where $\hat{A} \stackrel{def}{=} A \circ \iota_R$. That is, *TransformCond* specifies the subset of algebras having the left-commutative property, i.e. *LeftCommuting* specified above.

We now need to show the proof obligations arising as instances of axioms (F1) – (F6). For instance, the reductivity property (F2) is proven by structural induction on lists, and the proof of (F5) (which is the most complicated) is based on an auxiliary lemma showing

$$\frac{A \in \mathit{TransformCond} \quad x\ \mathit{mem}\ xs}{\mathit{Fold}\ A\ (x \# (\mathit{remove1}\ x\ xs)) = \mathit{Fold}\ A\ xs}$$

where *mem* denotes the membership test on lists and which can be shown by induction as well. All other proofs mainly comprise unfolding of definitions and case distinctions. Ultimately, we obtain the theory morphism *bag2* : *Fold-Param* $\longrightarrow$ *Bag* and the instantiation

instantiate-theory *Fold* **by-thymorph** *bag2*

which gives us the theory morphism $\mathit{bag2}^\sharp$: *Fold* $\longrightarrow$ *Bag*. Then, the central congruence property (11) for $\mathit{Fold}^{\mathsf{bag}}$ can be translated from *Fold* along $\mathit{bag2}^\sharp$. Based on this, we define the function *foldBag*:

$$\mathit{foldBag}\ f\ e \equiv \mathit{Fold}^{\mathsf{bag}}\ A \quad \text{where } A\ x \stackrel{def}{=} \begin{cases} f\ u\ v & \text{if } x = \iota_R(u, v) \\ e & \text{otherwise} \end{cases}$$

Altogether, we complete the development with a step constructing a theory morphism from *Bag-Spec* to the current development, corresponding to step (v) above. The emerging proof obligations, i.e. instances of bag axioms, can be now simply shown by unfolding the definitions (e.g. *foldBag*), and applying the congruence properties (e.g. (11)).

4.3 Implementing Finite Sets

Although the implementation of finite sets is considerably more complicated, it follows the same principle. The following development makes an intermediate step deriving the type of distinct lists, where any element occurs at most once.

Distinct lists. The theory *DList* of distinct lists starts with the definition of the function $Norm :: \alpha\ \mathsf{list} \Rightarrow \alpha\ \mathsf{list}$ by

$$\begin{aligned} Norm\ Nil &= Nil \\ Norm\ (x\#xs) &= x\#(removeAll\ x\ (Norm\ xs)) \end{aligned}$$

where $removeAll\ x\ xs$ removes all occurrences of x from the list xs. Let $\sim_{Norm}$ abbreviate $ker\ Norm$, i.e. the kernel relation of $Norm$. Then, the instantiation of *QuotientType* by the theory morphism, sending $\alpha\ \mathsf{T}$ to $\alpha\ \mathsf{list}$ and $\simeq$ to $\sim_{Norm}$, introduces the quotient type constructor dlist (using renaming $\mathsf{T}/{\simeq} \mapsto \mathsf{dlist}$), the corresponding extension function $\_^{\mathsf{dlist}} :: (\alpha\ \mathsf{list} \Rightarrow \beta) \Rightarrow \alpha\ \mathsf{dlist} \Rightarrow \beta$ and the congruence predicate $congruence_{\sim_{Norm}} :: (\alpha\ \mathsf{list} \Rightarrow \beta)\ \mathsf{set}$. It is now not difficult to show that for any $x :: \alpha$ the functions

1. $x\ mem\ \_$,
2. $class\text{-}of_{\sim_{Norm}} \circ (x\ \#\ \_)$, and
3. $class\text{-}of_{\sim_{Norm}} \circ (removeAll\ x)$

are in $congruence_{\sim_{Norm}}$. Let mem^D, put^D and get^D denote their respective extensions to $\alpha\ \mathsf{dlist}$. Moreover, let $empty^D \equiv [Nil]_{\sim_{Norm}}$. The definition of $Norm$ provides also another useful property:

$$xs \neq Nil \Longrightarrow xs \sim_{Norm} ys \Longrightarrow head\ xs = head\ ys$$

where $head$ is a function satisfying the equation $head\ (x\#xs) = x$. So, we can extend $head$ to $head^D :: \alpha\ \mathsf{dlist} \Rightarrow \alpha$ such that the proposition

$$xs \neq Nil \Longrightarrow head^D\ [xs]_{\sim_{Norm}} = head\ xs$$

is derivable. Based on this, we further have the following central decompositional property of distinct lists:

$$ds \neq empty^D \Longrightarrow ds = put^D\ h\ (get^D\ h\ ds) \quad \text{where } h \stackrel{def}{=} head^D\ ds$$

To derive a fold-hylomorphism for distinct lists from *foldr*, an application of the $\langle$*Fold-Param*, *Fold*$\rangle$ parametrisation is unnecessary. Instead, we can directly define

$$fold^D\ f\ e \equiv (foldr\ f\ e\ \circ\ Norm)^{\mathsf{list}/\sim_{Norm}}$$

and subsequently show

$$fold^D\ f\ e\ empty^D = e \tag{12}$$

$$\frac{\neg\ x\ mem^D\ ds}{fold^D\ f\ e\ (put^D\ x\ ds) = f\ x\ (fold^D\ f\ e\ ds)} \tag{13}$$

$$\frac{f \in LeftCommuting \quad x\ mem^D\ ds}{fold^D\ f\ e\ (put^D\ x\ (get^D\ x\ ds)) = fold^D\ f\ e\ ds} \tag{14}$$

These are the essential properties for the implementation of finite sets below.

The theory *FiniteSet*. The theory *FiniteSet* imports *DList* and defines the equivalence relation $\sim$ on distinct lists by $ds \sim ds' \equiv (\forall x.\, x\, mem^D\, ds = x\, mem^D\, ds')$. Thus, the theory morphism *fset1* : *QuotientType-Param* $\longrightarrow$ *FiniteSet*, sending $\alpha\,\mathsf{T}$ to $\alpha\,\mathsf{dlist}$ and $\simeq$ to $\sim$, provides the instantiation:

instantiate-theory *QuotientType* **by-thymorph** *fset1*
renames : $[\,\mathsf{T}/_{\simeq} \mapsto \mathsf{finite\text{-}set}]$

which gives us the new quotient type constructor $\mathsf{dlist}/_{\sim}$ as $\mathsf{finite\text{-}set}$ together with the extension function $\_^{\mathsf{finite\text{-}set}} :: (\alpha\,\mathsf{dlist} \Rightarrow \beta) \Rightarrow \alpha\,\mathsf{finite\text{-}set} \Rightarrow \beta$ and the congruence predicate $congruence_{\sim} :: (\alpha\,\mathsf{dlist} \Rightarrow \beta)\,\mathsf{set}$. Regarding the specification of finite sets, we can then prove the $\sim$-congruence properties of mem^D, put^D, and get^D:

1. $x\, mem^D\, \_ \in congruence_{\sim}$ for any x;
2. $(class\text{-}of_{\sim} \circ (put^D\, x)) \in congruence_{\sim}$ for any x;
3. $(class\text{-}of_{\sim} \circ (get^D\, x)) \in congruence_{\sim}$ for any x;

such that $(mem^D)^{\mathsf{finite\text{-}set}}$, $(put^D)^{\mathsf{finite\text{-}set}}$, and $(get^D)^{\mathsf{finite\text{-}set}}$ give us the implementations for the operations $\_ \Lsh \_$, $\_ \oplus \_$, and $\_ \ominus \_$ from *FiniteSet-Spec*, respectively.

We now turn to a derivation of *foldSet* from $fold^D$ using the parametrisation $\langle$*Fold-Param*, *Fold*$\rangle$. The formal type parameter $(\alpha, \beta)\,\mathbf{\Sigma}$ is mapped to $\mathbf{1} + \alpha \times \beta$. The parameter constants are mapped as follows:

1. Since the signature instantiation is the same as in *Bag*, the corresponding actions on relations and mappings do not change;
2. The coalgebra parameter c_{T} is instantiated by the function c_{dlist}, defined by

$$c_{\mathsf{dlist}}\; ds \equiv \begin{cases} \iota_L\, * & \text{if } ds = empty^D \\ \iota_R(head^D\; ds, get^D\; (head^D\; ds)\; ds) & \text{otherwise} \end{cases}$$

3. The hylomorphism *Fold* is given by the $fold^D$-function:

$$Fold\, A \;\equiv\; fold^D\; (\lambda x\, v.\; A(\iota_R(x, v)))\; A(\iota_L\, *)$$

4. The transformation is defined by

$$Transform\; u\; v \equiv \begin{cases} \iota_R(x, get^D\; x\; (put^D\; y\; ds')) & \text{if } u = \iota_R(x, ds), v = \iota_R(y, ds') \\ v & \text{otherwise} \end{cases}$$

5. Both conditions *FoldCond* and *TransformCond* are defined as in *Bag*.

The proofs of the emerging proof obligations are also similar to those for bags in Sect. 4.2. The proof of (F5) is again the most complicated and uses the properties (12), (13), and (14). Finally, the subsequent instantiation of the theory *Fold* gives the corresponding $\sim$-instance of the congruence property (11) for the extended function $Fold^{\mathsf{finite\text{-}set}}$: for any $A \in TransformCond$, i.e. for any algebra having

the left commutative property, the identity $Fold^{\mathsf{finite\text{-}set}}\ A\ [s]_\sim = Fold\ A\ s$ holds. Thus, we define the function *foldSet*:

$$foldSet\ f\ e \equiv Fold^{\mathsf{finite\text{-}set}}\ A \quad \text{where } A\ x \stackrel{def}{=} \begin{cases} f\ u\ v \text{ if } x = \iota_R(u,v) \\ e \quad \text{otherwise} \end{cases}$$

As the final step, the development is completed by constructing a theory morphism from the specification *FiniteSet-Spec* to the current development. The resulting proof obligations are now straightforward.

5 Conclusions

This paper has presented the formalisation of an abstract design tactic in Isabelle, which provides a way to define hylomorphisms on a quotient type. The design tactic has two parameter theories: first, the type and equivalence relation for the quotient, and second a functor representing a signature, a coalgebra and a transformation function, which providing the setting for a class of 'extensible' hylomorphisms, justified by Theorem 1. To apply the design tactic, concrete instantiations of the parameter theories have to be provided by giving instantiating theories and a morphism mapping the parameter theories. In our case, we have shown how to apply the design tactic for a systematical derivation of correct implementations of finite multisets and finite sets.

The formalisation presented here has used Isabelle; however, the development knowledge represented in the design tactic could be formalised in other theorem provers too, since it formalises conditions for folding over a quotiented type on an abstract level, and the constructions used in the formalisation can be found in most other theorem provers as well.

For future work, the tactic might be also further generalised: for example, we can capitalise on the fact that the type constructor Σ and two actions Σ^{Rel}, Σ^{Map} on relations and mappings form a *relator* [1], pointing to a possible formalisation already at the level of allegories, increasing the application area.

Further, [11] considers behavioural equivalence on algebras over the same signature w.r.t. a set *OBS* of observable types. From this point of view, the theories *Bag-Spec* and *FiniteSet-Spec* are data abstractions, since both specify classes of algebras, each closed under the behavioural equivalence where $OBS_{bags} \stackrel{def}{=} \{\mathsf{nat}\}$ and $OBS_{sets} \stackrel{def}{=} \{\mathsf{bool}\}$. Then the quotient tactic allows us to construct from algebras with lists as carrier, *Bag-Spec* and *FiniteSet-Spec* instances where the extensionality principle (axiom (S4)) additionally holds, introducing new quotient type. Future work includes examining further connections to the constructions in [11], like abstract and behaviour.

Acknowledgements. This research was supported by the German Research Foundation (DFG) under grants LU-707/2-1 and 2-2, and by the German Federal Ministry of Education and Research (BMBF) under grant 01 IM F02 A.

References

1. Bird, R., de Moor, O.: Algebra of Programing. Prentice Hall, Englewood Cliffs (1997)
2. Bortin, M., Johnsen, E.B., Lüth, C.: Structured formal development in Isabelle. Nordic Journal of Computing 13, 2–21 (2006)
3. Burstall, R.M., Goguen, J.A.: Putting theories together to make specifications. In: Proc. Fifth International Joint Conference on Artificial Intelligence IJCAI 1977, pp. 1045–1058 (1977)
4. Burstall, R.M., Goguen, J.A.: The semantics of CLEAR, a specification language. In: Bjorner, D. (ed.) Abstract Software Specifications. LNCS, vol. 86, pp. 292–332. Springer, Heidelberg (1980)
5. Chicli, L., Pottier, L., Simpson, C.: Mathematical quotients and quotient types in Coq. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 95–107. Springer, Heidelberg (2003)
6. Diaconescu, R., Futatsugi, K.: CafeOBJ Report. World Scientific, Singapore (1998)
7. Doornbos, H., Backhouse, R.C.: Induction and recursion on datatypes. In: Möller, B. (ed.) MPC 1995. LNCS, vol. 947, pp. 242–256. Springer, Heidelberg (1995)
8. Farmer, W.M., Guttman, J.D., Thayer, F.J.: Little theories. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 567–581. Springer, Heidelberg (1992)
9. Goguen, J.A.: A categorical manifesto. Tech. Rep. PRG-72, Oxford University Computing Laboratory, Programming Research Group, Oxford, England (1989)
10. Hofmann, M.: A simple model for quotient types. In: Dezani-Ciancaglini, M., Plotkin, G. (eds.) TLCA 1995. LNCS, vol. 902, pp. 216–234. Springer, Heidelberg (1995)
11. Hofmann, M., Sannella, D.: On behavioural abstraction and behavioural satisfaction in higher-order logic. Theoretical Computer Science 167, 3–45 (1996)
12. Homeier, P.V.: A design structure for higher order quotients. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 130–146. Springer, Heidelberg (2005)
13. Mossakowski, T., Autexier, S., Hutter, D.: Development graphs — proof management for structured specifications. Journal of Logic and Algebraic Programming 67(1-2), 114–145 (2006)
14. Mosses, P.D. (ed.): CASL Reference Manual. LNCS, vol. 2960. Springer, Heidelberg (2004)
15. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
16. Nogin, A.: Quotient types: A modular approach. In: Carreño, V.A., Muñoz, C.A., Tahar, S. (eds.) TPHOLs 2002. LNCS, vol. 2410, pp. 263–280. Springer, Heidelberg (2002)
17. Paulson, L.C.: Defining functions on equivalence classes. ACM Trans. Comput. Log. 7(4), 658–675 (2006)
18. Sannella, D., Burstall, R.: Structured theories in LCF. In: Protasi, M., Ausiello, G. (eds.) CAAP 1983. LNCS, vol. 159, pp. 377–391. Springer, Heidelberg (1983)
19. Slotosch, O.: Higher order quotients and their implementation in Isabelle/HOL. In: Gunter, E.L., Felty, A.P. (eds.) TPHOLs 1997. LNCS, vol. 1275, pp. 291–306. Springer, Heidelberg (1997)
20. Smith, D.R., Lowry, M.R.: Algorithm theories and design tactics. Science of Computer Programming 14, 305–321 (1990)
21. Srinivas, Y.V., Jullig, R.: Specware: Formal support for composing software. In: Möller, B. (ed.) MPC 1995. LNCS, vol. 947, Springer, Heidelberg (1995)