Embedding Algebraic Specifications in HOL

Dan Zhou[1] and Shiu-Kai Chin[2]

[1] Department of Computer Science and Engineering, Florida Atlantic University
dan@cse.fau.edu
[2] Department of Electrical Engineering and Computer Science, Syracuse University
chin@cat.syr.edu

Abstract. To build correct software systems in a systematic way, we need to integrate analysis with synthesis and support composition. Integrating analysis and synthesis gives us confidence in the system implementation. Composition facilitates reuse and is a crucial element of component-based software engineering. Logic, algebraic specification, and category theory provide a common ground for both synthesis and analysis. In this paper we outline these common elements of software system analysis and synthesis: specification, refinements and composition of specification and refinement. We define categories that are relevant to software engineering. We embed algebraic specifications as categories in HOL.

1 Introduction

Software systems are difficult to build correctly. The more complex the system, the more difficult to build it correctly. There is a great need for a systematic way to construct correct software systems. This systematic method should guarantee the correctness of an implementation. It should support component-based design and guarantee the correctness of an implementation if it is composed of components in a specific way.

There are two paths in software system engineering: a synthesis path, which takes a customer's requirement and builds a system that may or may not satisfy the requirement, and an analysis path, which checks an implementation against the system requirement. Traditionally, these two paths are not formally connected. By integrating analysis and synthesis we can construct software with high assurance. This integration is possible with higher-order logic.

One method of constructing systems uses algebraic specification, logic, and categories. Algebraic specification provides us with a way to describe abstract data types without committing to implementation details. There are different levels of specification abstractions. A lower-level specification has more features related to implementation. A specification refinement is a relation between two specifications, possible at different level of abstractions. Refinements move us closer to implementations. A specification sp_2 is a valid refinement of specification sp_1 if elements of sp_2 satisfy all the properties that elements of sp_1 satisfy with additional accidental (implementation-specific) properties.

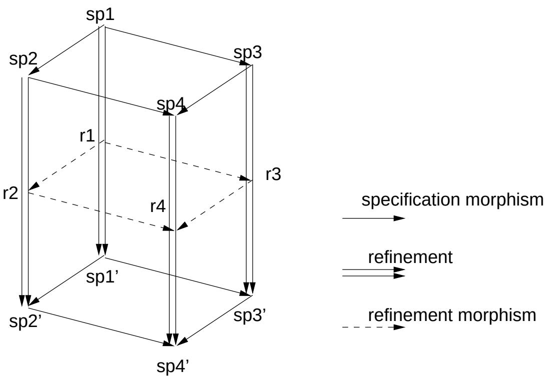


Fig. 1. Composition of refinements based on categories

We can develop a software system that is correct-by-construction based on algebraic specifications and stepwise refinements. A high-level system specification sp_1 is refined to implementation-level specification sp_n through a series of refinements. Each refinement step is so small that its correctness can be guaranteed. Specifications can be decomposed to and composed from smaller (component) specifications. A composed specification can be refined through composing refinements of its components. A specification can be refined to a low-level specification by sequentially composing intermediate refinements.

Categories provide the means to compose specifications and refinements such that the composed refinements are correct refinements of composed specifications. As illustrated in Figure 1, specification $sp4$ is the canonical composite specification that exhibits all the properties of both $sp2$ and $sp3$. Refinement $r4$ is the composite refinement of refinements $r1, r2,$ and $r3$. Category theory guarantees that $r4$ is a valid refinement from composite specification $sp4$ to $sp4'$ if we construct component refinements $r1, r2,$ and $r3$ in certain ways.

Logic acts as a bridge between analysis and synthesis. It provides a set of rules for reasoning about artifacts described in the logic. We can derive key properties of specifications, refinements, and composition operators once we have defined them within the calculus of higher-order logic.

Specware for specification refinement and HOL for formal analysis are examples of automated tools. Specware is based on algebraic specification, composition, categories and higher-order logic [5]. HOL, a higher-order logic theorem prover, provides automatic type definition and definitional extension to the logic [2]. To bridge the gap between the synthesis and analysis, we deeply embed the logic basic for synthesis into higher-order logic, namely, we deeply embed the algebraic specification into HOL. We have utilized Morris's embedding of category theory in HOL [3].

The purpose of embedding algebraic specification in HOL is to define precisely the operators relevant to software construction and to prove properties about them. We can accurately capture algebraic specification, specification refinements, and composition of specifications and refinements in HOL. This approach provides us with verified synthesis operators. The software output from synthesis tool is correct by construction. The embedding provides a basic for the correctness of the implementation.

To do deep embedding in HOL, we follow the well defined recipe for deeply embedding outlined in the works such as the one done by Camilleri and Melham [1]: define abstract syntax as types and assign meaning of terms in higher-order logic. We define algebras, terms, equations and specifications as HOL types and assign logic meanings to these types. Our contribution in this work is the embedding of operations related to refinements in HOL.

The rest of the paper describes the relevant constructs that need to be embedded in HOL and the status of our work. Section 2 briefly describes basic category theory concepts that are relevant to software construction. Section 3 gives an information description of signatures, algebras, and specifications. Specification refinements are described in Section 4. Sections 5-7 describe the embedding of signatures, algebras and algebra specifications in HOL.

2 Overview of Category Theory

Here we give an overview of category theory concepts that pertain to software development, such as category, diagram, pushout, and functor. A more complete introduction on category theory is given by Pierce [4].

2.1 Basic Concepts of Category Theory

Category A *category* $\mathcal{C}$ consists of:

- A collection of objects $O_{\mathcal{C}}$. We call each element in $O_{\mathcal{C}}$ a $\mathcal{C}$ object.
- A collection of arrows $A_{\mathcal{C}}$. Each arrow has a domain and a codomain that are objects. If f is an arrow whose domain and codomain are objects A and B, respectively, we write $f : A \to B$. We call each element in $A_{\mathcal{C}}$ a $\mathcal{C}$ arrow.
- For each $\mathcal{C}$ object A, there is an identity arrow $id_A : A \to A$.
- For each pair of $\mathcal{C}$ arrows $f : A \to B$ and $g : B \to C$, there is a composite arrow $g \circ f : A \to C$. The operator $\circ$ is called composition and satisfies the associativity properties:
  - For any $\mathcal{C}$ arrow $f : A \to B$, $f \circ id_A = f = id_B \circ f$.
  - For any $\mathcal{C}$ arrows $f : A \to B$, $g : B \to C$, and $h : C \to D$, $h \circ (g \circ f) = (h \circ g) \circ f$

A category $\mathcal{D}$ is a *subcategory* of category $\mathcal{C}$ if every $\mathcal{D}$ object is also a $\mathcal{C}$ object, every $\mathcal{D}$ arrow is also a $\mathcal{C}$ arrow, and identity arrows and composition arrows are defined to be the same as in $\mathcal{C}$. Category $\mathcal{D}$ becomes a *full sub-category* of $\mathcal{C}$ if all the $\mathcal{C}$ arrows between two $\mathcal{D}$ objects are also $\mathcal{D}$ arrows.

A *dual category* of category $\mathcal{C}$ is the category with the same objects and arrows as $\mathcal{C}$ with the directions of the arrows reversed. The dual category of $\mathcal{C}$ is written as $\mathcal{C}^{op}$.

Diagram Categories can be represented graphically using diagrams. A *diagram* $\mathcal{G}$ in category $\mathcal{C}$ is a directed multi-graph that is labeled consistently. That is, all the vertices of $\mathcal{G}$ are objects in $\mathcal{C}$, and all the edges of $\mathcal{G}$ are arrows in $\mathcal{C}$. Moreover, if e is an edge from A to B in $\mathcal{G}$, then $e : A \to B$ is an arrow in $\mathcal{C}$.

Equations in categories are represented using commutative diagrams. A diagram $\mathcal{G}$ in category $\mathcal{C}$ is *commutative* if all the multi-segment paths between two vertices A and B compose to the same arrow in $\mathcal{C}$.

Pushout, cocone, and colimit In categories, quotients of disjoint unions (sums) are represented using colimits. A pushout is a colimit of a specific diagram.

Pushouts provide a canonical way of composing objects in categories. A *pushout* of arrows $f : A \to B$, $g : A \to C$ is a pair of arrows $p : B \to D$ and $q : C \to D$ such that:

- $p \circ f = q \circ g$
- for all $p' : B \to D'$ and $q' : C \to D'$, there exists a unique arrow $u : D \to D'$ such that $p' = u \circ p$, $q' = u \circ q$.

A *cocone* of a diagram $\mathcal{D}$ in category $\mathcal{C}$ is a $\mathcal{C}$ object X together with the $\mathcal{C}$ arrows $f_i : D_i \to X$ for all objects $D_i \in \mathcal{D}$ such that, if $g : D_i \to D_j$ is an arrow in $\mathcal{D}$, then $f_i = f_j \circ g$. We denote the cocone as $\{f_i : D_i \to X \mid D_i \in \mathcal{D}\}$. Diagram $\mathcal{D}$ is called the *base* of the cocone.

A *colimit* of a diagram $\mathcal{D}$ is a cocone $\{f_i : D_i \to X \mid D_i \in \mathcal{D}\}$ such that for any other cocone $\{h_i : D_i \to X' \mid D_i \in \mathcal{D}\}$ there is a unique arrow $k : X \to X'$ in $\mathcal{C}$ such that, for all i, $h_i = k \circ f_i$.

Functor A *functor* is a pair of functions f_{obj} and f_{arr} between two categories $\mathcal{C}$ and $\mathcal{D}$ such that:

- For each $\mathcal{C}$ object A, $f_{obj}\ A$ is a $\mathcal{D}$ object.
- For each $\mathcal{C}$ arrow $g : A \to B$, $f_{arr}\ g$ is a $\mathcal{D}$ arrow from $f_{obj}\ A$ to $f_{obj}\ B$.
- For each $\mathcal{C}$ object A, $f_{arr}\ id_A = id_{f_{obj}\ A}$.
- For each pair of $\mathcal{C}$ arrows $g : A \to B$ and $h : B \to C$, $f_{arr}\ ((h \circ g) : A \to C) = (f_{arr}\ h \circ f_{arr}\ g) : f_{obj}\ A \to f_{obj}\ C$.

We use f to denote both f_{obj} and f_{arr}.

Category **Cat** has categories as objects and functors as arrows.

2.2 Category Theory in HOL

We based our work on Morris's definition of category theory in HOL [3]. Morris's work is quite similar to Agerholm's formalization of category theory [6]. The major difference is that Morris treats an arrow as an explicit triple with its domain and codomain objects, while Agerholm treats an arrow as a primitive entity. Morris's representation of arrows arises from his experience in proving a large number of elementary propositions of category theory.

The rest of the section briefly describes the relevant category theory definitions done by Morris.

In HOL, a category C is represented as a four-tuple with certain properties. A pre-category is a four-tuple

$$\begin{aligned}(O, A, Id, o) :& (\alpha \to bool) \times \\ & (\alpha \times \gamma \times \alpha \to bool) \times \\ & (\alpha \to \alpha \times \gamma \times \alpha) \times \\ & (\alpha \times \gamma \times \alpha \to \alpha \times \gamma \times \alpha \to \alpha \times \gamma \times \alpha)\end{aligned}$$

where

- Function O picks out C-objects from pre-objects of HOL type α.
- Function A picks out C-arrows from pre-arrows of type $\alpha \times \gamma \times \alpha$.
- Function Id constructs an identity arrow for a $\mathcal{C}$-object.
- Function $\circ$ constructs a composite arrow for two $\mathcal{C}$-arrows f and g if the domain of g equals to the codomain of f.

In HOL, an arrow a is represented as a triple $(d, f, c) : \alpha \times \gamma \times \alpha$, where d, f and c are the domain, the arrow itself, and the codomain of arrow a, respectively.

In Morris's treatment of category theory, whenever we are concerned only with a function's behavior over a certain domain, the behavior of the function outside the domain is forced to be of value ARB. Value ARB is based on the Hilbert operator ε, and ARB of any type τ is the term $\varepsilon : \tau.T$.

A predicate $isCat$ is defined to check if a four-tuple is a category. Any four-tuple that satisfies $isCat$ defines a category. A compound type $(\alpha, \gamma)cat$ is defined using a type-definition construct of HOL, where α is the type of pre-objects and $(\alpha \times \gamma \times \alpha)$ is the type of pre-arrows.

A detailed description of HOL functions can be found in Appendix A.

3 Overview of Algebraic Specifications

Algebraic specifications provide an implementation-independent way to describe abstract data types [7]. An algebraic specification contains the signature and characteristic properties of an abstract data type. The properties can be represented as equations in terms of the algebra. An algebraic specification provides a public interface for communicating with other specifications and hides the accidental features of an implementation.

3.1 Category for Signatures

A signature Σ is a pair (S, Ω) where S is a set of sorts and Ω is a set of function symbols. Each function symbol in Ω has a type $(s_1 \times s_2 \times s_3 \times \cdots \times s_n) \to s_0$, where $s_i \in S$ for all $i \in \{0, 1, \cdots, n\}$. A function symbol with type $\to s$ denotes a constant and is said to have type s.

The category **Sig** has signatures as objects and signature morphisms as arrows. A signature morphism between two signatures (S_1, Ω_1) and (S_2, Ω_2) is a pair of functions $f_s : S_1 \to S_2$ and $f_o : \Omega_1 \to \Omega_2$ such that the mapping between function symbols is consistent with that between sorts.

3.2 Category for Σ-Algebras

A Σ-algebra is a triple (Σ, A, I) where $\Sigma = (S, \Omega)$ is a signature, A is a set of sort-indexed carrier sets (assigning meanings to sorts), and I is a set of functions (assigning meaning to function symbols) such that functions have consistent type assignments with regard to carrier set assignments and function symbols:

$$\begin{aligned} A &= \{A_s \mid s \in S\} \\ I &= \{I_\rho \mid \rho \in \Omega\},\ I_\rho : (A_{s_1} \times A_{s_2} \cdots \times A_{s_n} \to A_s) \end{aligned}$$

Category $\mathbf{Alg}_\Sigma$ is a category whose objects are Σ-algebras and whose arrows are Σ-homomorphisms. A Σ-homomorphism between (Σ, A, I) and (Σ, A', I') is a suite of functions $h_1 : A_{s_1} \to A'_{s_1}, h_2 : A_{s_2} \to A'_{s_2}, \cdots, h_n : A_{s_n} \to A'_{s_n}$ such that

$$h(I_\rho(v_1, v_2, ..., v_n)) = I'_\rho(h\ v_1, h\ v_2, \ldots, h\ v_n)$$

We use $h : A \to A'$ to denote $h_i, i \in \{1, 2, \cdots, n\}$.

3.3 Terms of Σ-Algebra

We can extend a signature $\Sigma = (S, \Omega)$ with an S-indexed family of free variables $\Gamma = \{\Gamma_s \mid s \in S\}$ to $\Sigma(\Gamma) = (S, \Omega')$. Set Ω' is extended from Ω, where every element of Γ_s becomes a constant in Ω' with type s.

Terms of $\Sigma(\Gamma)$ are defined inductively:

- If variable x is of type s, then x is a term of type s.
- If t_i is a term of type s_i, for $i \in \{1, 2, \cdots, n\}$, and ρ is a function symbol of type $(s_1 \times s_2 \times \cdots \times s_n \to s)$, then $\rho(t1, t2, \ldots, tn)$ is a term of type s.

If we allow terms of function types, we can define $\Sigma(\Gamma)$ terms indirectly. We call terms of function types $\Sigma_{pre}(\Gamma)$ terms and define them inductively:

- If variable x is of type s, then x is a term of type $\to s$ or type s.
- If ρ is a function symbol of type $(s_1 \times s_2 \times \cdots \times s_n \to s)$, then ρ is a term of type $(s_1 \times s_2 \times \cdots \times s_n \to s)$.

– If t_1 is a term of type $(s_1 \times s_2 \times \ldots \times s_n \rightarrow s)$ and t_2 is a term of type s_1, then $(t_1\ t_2)$ is a term of type $(s_2 \times \ldots \times s_n \rightarrow s)$.

A $\Sigma(\Gamma)$ term of type s is a $\Sigma_{pre}(\Gamma)$ term of type $\rightarrow s$.

A Σ-equation is a pair of $\Sigma(\Gamma)$ terms (t_1, t_2) where both t_1 and t_2 have type s. It is written as $t_1 =_s t_2$.

3.4 Category for Algebraic Specifications

An algebraic specification is a pair (Σ, E) where Σ is a signature and E is a set of Σ-equations that are axioms and theorems of the specification. Algebraic specifications can be used to model computer systems, where Σ is the interface of the system and E is the desired system property.

A Σ-algebra (Σ, A, I) is a model of a specification (Σ, E) if (Σ, A, I) satisfies all the equations in E.

Category **Spec** is a category of algebraic specifications whose objects are specifications and whose arrows are specification arrows. A specification arrow f between two specifications (Σ, E) and (Σ', E') is a signature arrow between Σ and Σ' that preserves theorems: any axiom (i.e., Σ-equation) in E becomes either an axiom in E' or a theorem that is derivable from the axioms in E'. In other words, a specification arrow (also called a specification morphism) finds an image in Σ' for every element in Σ that preserves the axioms in E.

The categorical construct pushout is important in composing specifications. In particular, a pushout in category **Spec** gives a canonical specification that exhibits all the properties of the component specifications.

4 Overview of Specification Refinements

In this section we informally define categories and operations related to refinements. The concepts of refinements and refinement operations presented here are based on SPECWARE [5]. We plan to eventually embed this part into HOL.

Refinement of a problem specification (or source of higher-level abstraction) to a solution specification (or target of lower-level abstraction) is a principal construct in the synthesis path of software development based on algebraic specifications. When combined with category theory constructs for composition, the refinement of specifications can be defined in terms of the refinements of other specifications. Refinements can be composed through sequential compositions and parallel compositions of refinements.

4.1 Definitional Extension

The basic refinement construct in SPECWARE is an *interpretation. Definitional extensions* are the bases for interpretations. An algebraic specification sp_2 is a definitional extension of specification sp_1 if they have the same set of models.

More rigorously, a definitional extension can be defined in terms of a functor between the categories **Spec** and $\mathbf{Cat}^{op}$.

Let $Mod : \mathbf{Spec} \rightarrow \mathbf{Cat}^{op}$ be a functor such that (as illustrated in Figure 2):

1. For any specification (Σ, E), define
   $Mod\ (\Sigma, E) = \mathbf{Alg}(\Sigma, E)$
   where $\mathbf{Alg}(\Sigma, E)$ is a full subcategory of the category of $\Sigma-$algebras in which each $\Sigma-$algebra satisfies E
2. For any specification arrow $f : (\Sigma, E) \rightarrow (\Sigma', E')$, where
   $\Sigma = (S, \Omega)$ and $\Sigma' = (S', \Omega')$
   define
   $Mod\ f : \mathbf{Alg}(\Sigma', E') \rightarrow_{\mathbf{Cat}} \mathbf{Alg}(\Sigma, E)$
   as an arrow in category $\mathbf{Cat}$ such that (as illustrated in Figure 3)
   2.1. For any algebra (Σ', A', I') in category $\mathbf{Alg}(\Sigma', E')$, define
   $Mod\ f\ (\Sigma', A', I') = (\Sigma, A, I)$ as an object in category $\mathbf{Alg}(\Sigma, E)$
   where A and I are defined as follows:
   for each $s \in S, A_s = A'_{(f\ s)}$,
   for each $\rho \in \Omega, I_\rho = I'_{(f\ \rho)}$
   2.2. For any $\Sigma-$homomorphism $h' : (\Sigma', A'_1, I'_1) \rightarrow (\Sigma', A'_2, I'_2)$, define
   $Mod\ f\ h' = h : (\Sigma, A_1, I_1) \rightarrow (\Sigma, A_2, I_2)$
   where
   $(\Sigma, A_1, I_1) = Mod\ f\ (\Sigma', A'_1, I'_1)$ and
   $(\Sigma, A_2, I_2) = Mod\ f\ (\Sigma', A'_2, I'_2)$ and
   for each $s \in S$ and each $v \in A_{1s},\ v \in A'_{1(f\ s)}$
   and h is defined as follows:
   $h\ (v : s) = h'\ (v : f\ s)$

A definitional extension is a specification arrow f such that $(Mod\ f)$ is an isomorphism between two categories $Mod(\Sigma, E)$ and $Mod(\Sigma', E')$. If $f : sp_1 \Rightarrow sp_2$ is a definitional extension, we also say sp_2 is a definitional extension of sp_1.

4.2 Interpretations

Interpretation are extensions of specification morphisms between two specifications. A specification morphism provides an image in the target for every element in the source. An interpretation maps each element in the source to an element that is either in the target or is defined in terms of the elements in the target.

An interpretation S-*to*-T between two specifications $S = (\Sigma, E)$ and $T = (\Sigma', E')$ is a triple (S-*as*-T, s, t), where S-*as*-T is an algebraic specification, s is a specification morphism from S to S-*as*-T, and t is a definitional extension from T to S-*as*-T. Here we follow the naming convention in SPECWARE: specification S-*as*-T is known as mediator specification, or mediator of the interpretation, s

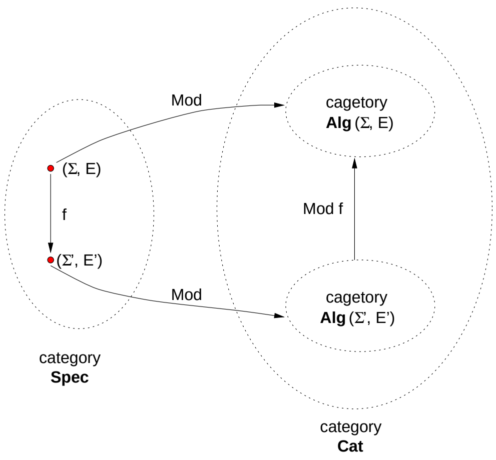


Fig. 2. Functor *Mod* between categories **Spec** and $\mathbf{Cat}^{op}$

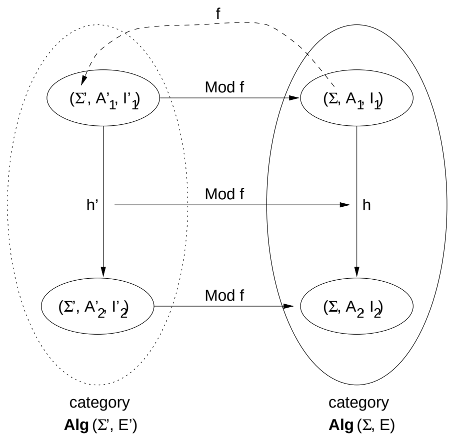


Fig. 3. Functor $(Mod\ f)$ between categories $\mathbf{Alg}(\Sigma', E')$ and $\mathbf{Alg}(\Sigma, E)$

as source morphism and t as target morphism. We use *S-to-T:* $S \Rightarrow T$ to denote an interpretation from S to T.

A more relaxed relation between two specifications is an *interpretation scheme*. An interpretation scheme (*ip-scheme*) has the same structure as an interpretation, except that the target morphism need not be a definitional extension.

4.3 Sequential Compositions of Refinements

We can implement a high-level specification by an intermediate-level specification, and implement the intermediate-level specification by a low-level specification. By composing these two refinements sequentially, we arrive at a low-level specification (implementation) for the high-level specification.

Categorically speaking, sequential compositions of refinements are the composition of arrows in category **Ref** for refinements.

Category **Ref** is a category of specifications (as objects) and interpretations (as arrows). The identity arrow for an object *sp* is the identity interpretation $id_{sp} : sp \Rightarrow sp$, where the mediator is *sp*, and the source-to-mediator morphism and target-to-mediator morphism are both identity specification morphisms.

As illustrated in Figure 4, the composition operation in **Ref** is defined as follows. The interpretation *R-to-T:* $R \Rightarrow T$ is the composition of *R-to-S:* $R \Rightarrow S$ and *S-to-T:* $S \Rightarrow T$. Interpretation *R-to-S* is the triple (*R-as-S*, *r1*, *s1*) and interpretation *S-to-T* is the triple (*S-as-T*, *s2*, *t1*). Composite interpretation *R-to-T* is the triple (*R-as-S-as-T*, *r3*, *t3*), where *R-as-S-as-T*, *r3*, and *t3* are all defined in terms of objects and arrows in the category **Spec**. Specification *R-as-S-as-T* and specification morphisms *r2* and *t2* are the pushout object and arrows of the diagram containing morphisms *s1* and *s2*, arrow *r3* is the sequential composition of *r1* and *r2*, and arrow *t3* is the sequential composition of *t1* and *t2*.

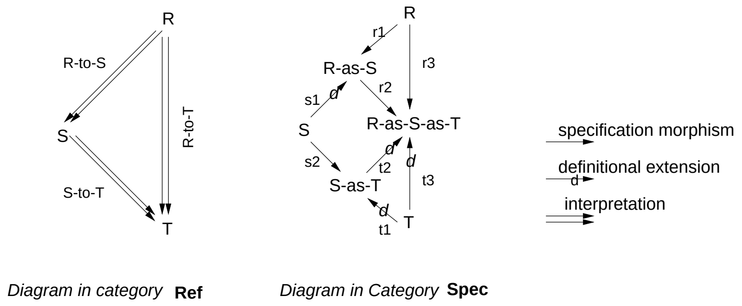


Fig. 4. Sequential composition of interpretations

4.4 Parallel Composition of Refinements

Category theory construct pushouts and colimits form the basis for composing both specifications and refinements. Just as a specification can be composed through a pushout operation in category **Spec**, a refinement can be composed through a pushout operation in the category of ip-schemes—category **Ip-Scheme**. This composition of refinements is called the parallel composition of refinements.

Category **IpScheme** has ip-schemes as objects and ip-scheme morphisms as arrows. An ip-scheme morphism from one ip-scheme $(A\text{-}as\text{-}B, a, b) : A \Rightarrow B$ to another ip-scheme $(S\text{-}as\text{-}T, s, t) : S \Rightarrow T$ is a triple of specification morphisms: $f : A \to S, g : A\text{-}as\text{-}B \to S\text{-}as\text{-}T,$ and $h : B \to T$ such that

$$g \circ a = s \circ f \text{ and } g \circ b = t \circ h$$

Identity arrow for any object $(S\text{-}as\text{-}T, s, t) : S \Rightarrow T$ is triple $(id_S, id_{S\text{-}as\text{-}T}, id_T)$. The composition operation is defined as the component-wise composition of specification morphisms:

$$\begin{aligned}&(f : A \to C,\ g : A\text{-}as\text{-}B \to C\text{-}as\text{-}D,\ h : B \to D) \circ_{\mathbf{IpScheme}}\\ &\quad (f' : C \to S,\ g' : C\text{-}as\text{-}D \to S\text{-}as\text{-}T,\ h' : D \to T) =_{\mathbf{def}}\\ &(f' \circ_{\mathbf{Spec}} f : A \to S,\ g' \circ_{\mathbf{Spec}} g : A\text{-}as\text{-}B \to S\text{-}as\text{-}T,\ h' \circ_{\mathbf{Spec}} h : B \to T)\end{aligned}$$

Another category related to refinements is the category **Interp**. Category **Interp** has interpretations as objects and is a full subcategory of category the **IpScheme**.

If a specification *sp* is composed through pushout constructs in the category **Spec**, we can construct a refinement from *sp* through pushout constructs in the category **IpScheme**, under the condition that the ip-schemes from the component specifications of *sp* satisfy certain constraints. This composed refinement is called a parallel composition of refinements from component specifications.

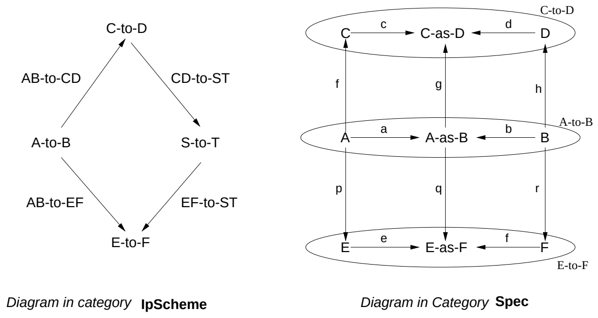
Parallel compositions of ip-schemes are the colimits of diagrams in category **IpScheme**, as shown in Figure 5(a). A colimit of a diagram in category **IpScheme** exists only when the corresponding component diagram in category **Spec** (as shown in Figure 5(a)) commutes. The colimit object $S\text{-}to\text{-}T$ is triple $(S\text{-}as\text{-}T,\ s : S \to S\text{-}as\text{-}T,\ t : T \to S\text{-}as\text{-}T)$ where S, s, and t are defined as follows. As illustrated in Figure 5(b), specification S and specification morphisms f' and p' are the pushout object and morphisms of the diagram containing morphisms f and p; specification $S\text{-}as\text{-}T$ and specification morphisms g' and q' are the pushout object and morphisms of the diagram containing morphisms g and q; and specification T and specification morphisms h' and r' are the pushout object and morphisms of the diagram containing morphisms h and r. As illustrated in Figure 5(c), morphisms s and q' are the pushouts of the diagram containing c and p'; and morphisms d and q' are the pushouts of the diagram containing r' and t.

When we take a colimit of a diagram containing interpretations and ip-schemes, the resulting ip-scheme may be an interpretation.

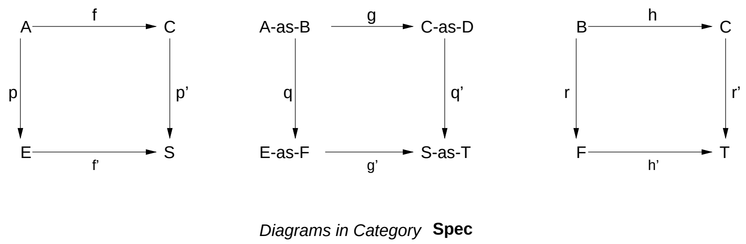
4.5 Refinement of Structured Specifications

If a specification R is a definitional extension of a specification S, then we can extend an interpretation of S into an interpretation of R.

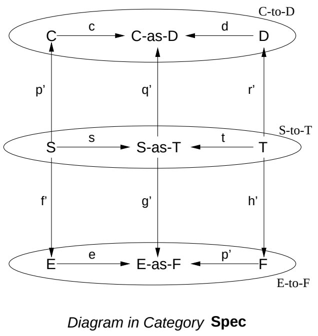
As shown in Figure 6, specification morphisms m and n are pushout arrows of f and g. Arrow p is the composition of h and m. Arrows with d represent definitional extension. Because definitional extensions are preserved on composition



(a)



(b)



(c)

Fig. 5. Parallel composition of interpretations

and pushout, arrow p is a definitional extension. Therefore the triple $R\text{-}as\text{-}T$, n and p make up an interpretation from R to T.

The colimit of a diagram of interpretations yields an interpretation from a colimit of the interpretations sources to a colimit of the interpretation targets.

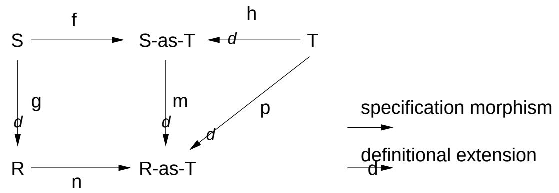


Fig. 6. Interpretations of definitional extensions

For example, as illustrated in Figure 7(a), suppose specification D is the colimit of diagram consisting specification morphisms m and n. To refine D, we make interpretations *A-to-M:* $A \Rightarrow M$, *B-to-N:* $B \Rightarrow N$, and *C-to-P:* $C \Rightarrow P$ in such a way that the diagram shown Figure 7(b) (in category **Spec**) commutes. We then take colimit of diagram of interpretations consisting ip-scheme morphisms f and g, and arrive at an interpretation *D-to-Q* for D.

5 Embedding Signatures in HOL

More detailed description of HOL functions can be found in Appendix B.

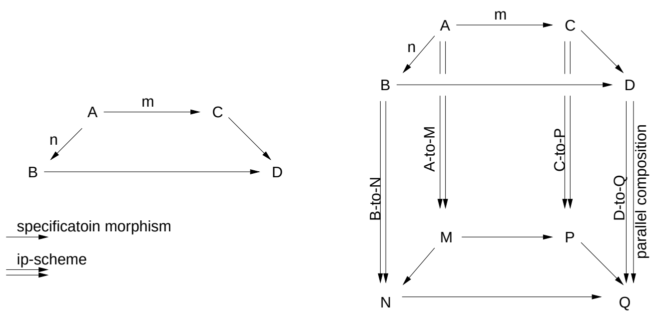
5.1 Signature as a HOL type

We define sorts and function symbols to be of base types *sort* and *operator* in HOL. The use of new base types gives us as much generality as with a type variable, because a signature is just a set of symbols.

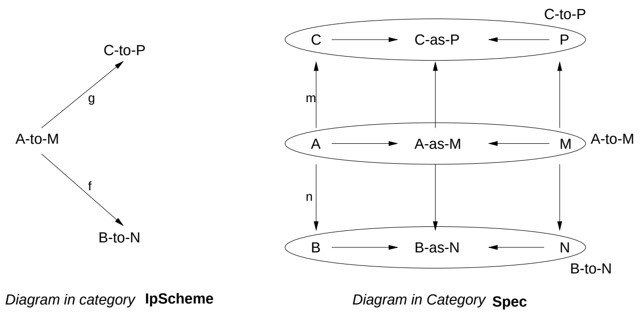
For a signature $\Sigma = (S, \Omega)$, S is represented in HOL as a set whose elements are of type *sort*. Based on the observation that every function symbol has an input type and an output type, Ω is represented in HOL as a set of triples $(\rho, sl, s) : \mathit{operator} \times \mathit{sort\ list} \times \mathit{sort}$, where ρ is a function symbol, sl is the type of input argument to ρ, and s is the return type of ρ. The function symbol's input type is a list of sorts. A constant c in Ω is represented by a triple $(c, [\,], s)$ where $[\,]$ is the empty list; c is said to be of type s. The use of triples to represent elements of set Ω allows us to overload functions symbols. Elements (which we shall call **function-names**) in Ω can be treated as primitives. Functions

$$rho : \mathit{operator} \times \mathit{sort\ list} \times \mathit{sort} \rightarrow \mathit{operator}$$
$$arg : \mathit{operator} \times \mathit{sort\ list} \times \mathit{sort} \rightarrow \mathit{sort\ list}$$
$$ret : \mathit{operator} \times \mathit{sort\ list} \times \mathit{sort} \rightarrow \mathit{sort}$$

obtain the function symbol, the argument type, and the return type of a function name.



(a)



(b)

Fig. 7. Colimit refinement

A pair (S, Ω) represents a signature if the input and output types of all function-names in Ω are restricted to the sorts in S. Predicate $isSig : sort\ set \times (operator \times sort\ list \times sort)set \rightarrow bool$ defines the subset of pairs that are valid representations of a type sig for signature.

5.2 Signature as a category

An arrow in category **Sig** is represented as a triple, and the second element as a pair:

$$
\begin{array}{l}
(d, (f_s, f_o), c) : \\
sig \times \\
((sort \rightarrow sort) \times \\
\quad ((operator \times sort\ list \times sort) \rightarrow (operator \times sort\ list \times sort))) \times \\
sig
\end{array}
$$

Predicate $sigA$ in HOL identifies the signature morphisms from pre-arrow triples $(d, (fs, fo), c)$. The identity arrow for a signature (S, Ω) is a pair of identity functions $id_S : S \to S$ and $id_\Omega : \Omega \to \Omega$. Composition of two arrows m and n is defined as the pairwise composition of their components.

We have proved that the so-defined category Sig is indeed a category.

6 Embedding Σ-algebra in HOL

More detailed description of HOL functions can be found in Appendix C.

6.1 Algebra as a HOL type

We use a new base-type $value$ in HOL to represent the values that are in a carrier set. In HOL, the set of sort-indexed carrier sets of a Σ-algebra is represented as a set of pairs $(v, s) : value \times sort$, where v is a value and s is the type of the value. This representation allows us to overload symbols for values. The set of functions of a Σ-algebra is represented as a set of functions, each taking a list of values to a value.

We use the triple (Σ, A, I) to represent a Σ-algebra in HOL, where A is a HOL function that constructs a carrier set, given a sort of the signature Σ, and I is a HOL function that maps a function-name to a function:

$$A : sort \to value\ set$$
$$I : (operator \times sort\ list \times sort) \to (value\ list \to value)$$

We call A the carrier-sets assignment function and I the function-name assignment function.

A pre-algebra triple (Σ, A, I) is a Σ-algebra if the mapping from function-names to functions is consistent with the mapping from sorts to carrier sets. Predicate $isAlg$ defines the subset of triples that are valid representations of type alg for algebras. We define type alg for algebras using HOL's type-definition construct.

6.2 Σ-Algebra as a category

Each signature Σ uniquely defines a category $\mathbf{Alg}_\Sigma$. We have defined category $\mathbf{Alg}_\Sigma$ in such way that the signature Σ is taken as a parameter.

An arrow in category $\mathbf{Alg}_\Sigma$ is of type $alg \times (sort \to value \to value) \times alg$.

The predicate $sigmaAlgA$ picks out $\mathbf{Alg}_\Sigma$ arrow from pre-arrows. The identity arrow in category $\mathbf{Alg}_\Sigma$ is the function $(\lambda s.I : value \to value)$. The composition of arrows (a, f, b) and (b, g, c) is the composition of functions $(a, \lambda s.(g\ s) \circ (f\ s), c)$.

6.3 Terms of Σ-Algebra

To define terms of Σ-algebra in HOL, we introduce a new base type *variable* to represent free variables. In HOL, the set of free variables is represented as a set of pairs $(x, s) : \mathit{variable} \times \mathit{sort}$ where x is the symbol of a free variable and $s \in S$ is the type of that variable. A variable assignment assigns types to free variable symbols and must be consistent with the set of sorts, for it to be a valid assignment.

We first define a HOL-type *ppreT* as a pre-$\Sigma_{pre}(\Gamma)$ term. Pre-$\Sigma_{pre}(\Gamma)$ terms are represented as recursive-type *ppreT* defined in HOL using a recursive-type definition package. A pre-$\Sigma_{pre}(\Gamma)$ term contains the type information of the term.

```
ppreT ::= Leafv (v, s) |          (a variable)
          Leafo (op, sl, s) |     (a function symbol)
          Comb ppreT ppreT;
```

In HOL, this syntax is represented by recursive-type *ppreT* with three constructors:

$$\begin{aligned} Leafv &: (\mathit{variable} \times \mathit{sort}) \rightarrow \mathit{ppreT} \\ Leafo &: (\mathit{operator} \times \mathit{sort\ list} \times \mathit{sort}) \rightarrow \mathit{ppreT} \\ Comb &: \mathit{ppreT} \rightarrow \mathit{ppreT} \rightarrow \mathit{ppreT} \end{aligned}$$

The terms of $\Sigma_{pre}(\Gamma)$ can now be defined inductively:

1. If x is a variable of type s, then $Leafv\ (x, s)$ is a $\Sigma_{pre}(\Gamma)$ term of type $([\,], s)$.
2. If op is a function symbol of type (sl, s), then $Leafo\ (op, sl, s)$ is a $\Sigma_{pre}(\Gamma)$ term of type (sl, s).
3. If $t_1 : (v_1, s_1)$ and $t_2 : ([\,], s_2)$ are terms of $\Sigma_{pre}(\Gamma)$ and $s_2 = HD\ v_2$, then $Comb\ t_1\ t_2$ is a $\Sigma_{pre}(\Gamma)$ term of type $(TL\ v_1, s_1)$.

The induction is formalized as follows:

$$\frac{}{isPreSigmaTerm\ sigma\ gamma\ ([\,], s)(Leafv\ (x, s))}$$

$$\frac{}{isPreSigmaTerm\ sigma\ gamma\ (sl, s)(Leafo\ (op, sl, s))}$$

$$\frac{\begin{array}{c} isPreSigmaTerm\ sigma\ gamma\ (v_1, s_1)t_1, \\ isPreSigmaTerm\ sigma\ gamma\ ([\,], s_2)t_2 \end{array}}{isPreSigmaTerm\ sigma\ gamma\ (TL\ v_1, s_1)(Comb\ t_1\ t_2)}\ s_2 = HD\ v_1$$

A $\Sigma(\Gamma)$-term of type s is just a $\Sigma_{pre}(\Gamma)$ term whose type is $([\,], s)$. $\Sigma(\Gamma)$-terms are represented as four-tuples $(\Sigma, \Gamma, t, s) : sig \times (\mathit{variable} \times \mathit{sort})set \times \mathit{ppreT} \times \mathit{sort}$. Predicate $isSigmaTerm$ defines the subset of the four-tuples that are valid representations of HOL-type $sigmaTm$ for Σ-terms.

A Σ-equation is a pair of terms $(t_1, t_2) : sigmaTm \times sigmaTm$ where t_1 and t_2 are of the same type. Predicate $isSigmaEq$ selects Σ-equations from pre-equation pairs. It defines the subset of pre-equations that are valid representation of HOL-type $sigmaEq$ for Σ-equations.

7 Embedding algebraic specifications in HOL

More detailed description of HOL functions can be found in Appendix D.

7.1 Specification as a HOL type

In HOL, specifications are represented by a triple $(\Sigma, \Gamma, E) : sig \times (variable \times sort)\ set \times sigmaEq\ set$ where Σ is a signature, Γ is a variable assignment, and E is a set of Σ-equations.

Predicate *isSpec* defines the subset of pre-specification triples that are valid representations of the HOL-type *spec* for algebraic specifications.

7.2 Algebraic specifications as a category

In HOL, a specification arrow is a triple of functions $f_s : S_1 \to S_2$, $f_o : \Omega_1 \to \Omega_2$ and $f_v : \Gamma_1 \to \Gamma_2$. The pair (f_s, f_o) is a signature arrow, and f_v is a mapping between two variable assignments of the specifications. Mapping f_v is necessary at this stage to avoid the complex α-conversion of terms.

A specification arrow is represented as a triple:

$$
\begin{aligned}
&(d, (fs, fo, fv), c) : \\
&spec \times \\
&((sort \to sort) \times \\
&\quad ((operator \times sort\ list \times sort) \to (operator \times sort\ list \times sort)) \\
&\quad ((variable \times sort) \to (variable \times sort))) \times \\
&spec
\end{aligned}
$$

The signature arrow $(f_s, f_o) : (S, \Omega) \to (S', \Omega')$ and variables mapping $f_v : \Gamma \to \Gamma'$ decide how to transform the Σ-equations of one specification to the Σ-equations of another specification. First, terms of pre-$\Sigma_{pre}(\Gamma)$ are transformed to those of pre-$\Sigma'_{pre}(\Gamma')$ recursively:

1. A term of form $Leafv\ (x, s)$ in pre-$\Sigma_{pre}(\Gamma)$ is transformed to $Leafv\ (fv\ (x, s))$.
2. A term of form $Leafo\ (op, sl, s)$ in pre-$\Sigma_{pre}(\Gamma)$ is transformed to $Leafo\ (fo\ (op, sl, s))$.
3. A term of form $Comb\, t1\, t2$ is transformed to $Comb\ (transPPT\ t1)\ (transPPT\ t2)$.

A $\Sigma(\Gamma)$-terms t of type s is transformed to term $t' = tranPPT\ t$ of type $(f_s\ s)$. A Σ-equation $t_1 =_s t_2$ is transformed to $t'_1 =_{s'} t'_2$, where $t'_1 = transPPT\ t_1$, $t'_2 = transPPT\ t_2$ and $s' = (f_s\ s)$.

We use a constant function $thmsDerivable : sigmaEq\ set \to sigmaEq\ set$ that constructs a theory from a set of axioms. We can then define specification arrows in HOL. Predicate *inGammaSpec* defines the elements that are in the variable-assignment set Γ of a specification. Predicate *specA* picks out the specification arrows from pre-arrow triples. The identity arrow of object (Σ, E) in category **Spec** is that of object Σ in category **Sig**. The composite arrow in category **Spec** is defined the same way as in category **Sig**.

8 Conclusion

We have defined signatures, Σ-algebras and algebraic specifications in HOL. The definitions allow the overloading of functions symbols and variable symbols. The definitions on Σ-algebra take signature Σ as a parameter. The definitions on algebraic specifications take variable assignment Γ as a parameter. The parameterized definitions allow reuse.

Currently, axioms and theorems in a specification contain only Σ-equations. This can be extended to contain connectives such as $\wedge$ and $\neg$ for conjunction and negation.

We have also defined categories **Sig**, $\mathbf{Alg}_{\Sigma}$ and **Spec**. We have proved **Sig** as defined is indeed a category.

In the definition of category **Spec**, a specification arrow contains mapping between two variable assignments of the specifications. If we add the facility to do α-conversion of terms, then the mapping between variables will be unnecessary.

We still need to prove that the Σ-algebras and algebraic specifications as defined are categories.

The goal of our research is to reason about refinements in HOL. To fulfill that goal, we need to formalize concepts related to refinement, which are based on the algebraic specifications.

References

1. J. Camilleri and T. Melham. Reasoning with Inductively Defined Relations in the HOL Theorem Prover. Technical report, University of Cambridge Computer Laboratory, August 1992.
2. M.J.C. Gordon. A Proof Generating System for Higher-Order Logic. In G. Birtwistle and P. A. Subramanyam, editors, *VLSI specification, verification and synthesis*. Kluwer, 1987.
3. Lockwood Morris. Interim Partial Description of a Representation for Categories in HOL. Communicated through private channel, 1998.
4. Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. The MIT Press, Cambridge, Massachusetts, 1991.
5. Yellamraju V. Srinivas and Richard Jullig. Specware: Formal Support for Composing Software. In *Conference on Mathematics of Program Construction*, Kloster Irsee, Germany, July 1995.
6. Sten Agerholm. Experiments in formalizing basic category theory in higher order logic and set theory. *http://www.cs.chalmers.se/ ilya/FMC/*, 1995.
7. Martin Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B, Formal Models and Semantics*, pages 675–788. Elsevier/The MIT Press, 1990.

A Category theory in HOL

This section briefly describes the relevant category theory definitions done by Morris.

In HOL, a category C is represented as a four-tuple with certain properties. A pre-category is a four-tuple

$$\begin{aligned}(O, A, Id, o) :& (\alpha \to \mathit{bool}) \times \\ & (\alpha \times \gamma \times \alpha \to \mathit{bool}) \times \\ & (\alpha \to \alpha \times \gamma \times \alpha) \times \\ & (\alpha \times \gamma \times \alpha \to \alpha \times \gamma \times \alpha \to \alpha \times \gamma \times \alpha)\end{aligned}$$

where

- Function O picks out C-objects from pre-objects of HOL type α.
- Function A picks out C-arrows from pre-arrows of type $\alpha \times \gamma \times \alpha$.
- Function Id constructs an identity arrow for a $\mathcal{C}$-object.
- Function $\circ$ constructs a composite arrow for two $\mathcal{C}$-arrows f and g if the domain of g equals to the codomain of f.

In HOL, an arrow a is represented as a triple $(d, f, c) : \alpha \times \gamma \times \alpha$, where d, f and c are the domain, the arrow itself, and the codomain of arrow a, respectively. The accessor functions *dom* and *cod* return the domain and codomain of a given arrow. The property *composable* asserts that two arrows of a given category are composable, and property *cpsl* asserts that two triples are arrows of a certain category and that they are composable.

```
dom ⊢def  ∀d m c. dom (d,m,c) = d
cod ⊢def  ∀d m c. cod (d,m,c) = c
composable ⊢def  ∀f g. composable f g = (dom f = cod g)
cpsl ⊢def  ∀A f g. cpsl A f g = A f ∧ A g ∧ composable f g
```

In Morris's treatment of category theory, whenever we are concerned only with a function's behavior over a certain domain, the behavior of the function outside the domain is forced to be of value ARB. Value ARB is based on the Hilbert operator ε, and ARB of any type τ is the term $\varepsilon : \tau.T$. Two definitions explore this idea. The Predicate *isRestr* checks that the value of function f outside the truth-set of predicate P is ARB, and $isRestr2$ checks that the value of a curried function of two arguments g outside the truth-set of predicate Q is ARB.

```
ARB = εx:*. T
isRestr ⊢def  ∀P f. isRestr P f = f = (λx ::P. f x)
isRestr2 ⊢def  ∀Q g. isRestr2 Q g = g = (λx. λy ::(Q x). g x y)
```

A predicate $isCat$ is defined to check if a four-tuple is a category.

```
isCat
⊢def  ∀O A id oo.
      isCat (O,A,id,oo) =
      (∀f ::A. O (dom f) ∧ O (cod f)) ∧
      isRestr2 (cpsl A) oo ∧
      (∀f g ::A. composable f g ⊃  A (oo f g)) ∧
      (∀f g ::A.
        composable f g ⊃
        (dom (oo f g) = dom g) ∧ (cod (oo f g) = cod f)) ∧
      (∀f g h ::A.
        composable f g ∧ composable g h ⊃
        (oo (oo f g) h = oo f (oo g h))) ∧
      isRestr O id ∧
      (∀a ::O. A (id a)) ∧
      (∀a ::O. (dom (id a) = a) ∧ (cod (id a) = a)) ∧
      (∀a ::O.
        (∀f ::A. (dom f = a) ⊃  (oo f (id a) = f)) ∧
        (∀g ::A. (cod g = a) ⊃  (oo (id a) g = g)))
```

Any four-tuple that satisfies *isCat* defines a category. A compound type $(\alpha, \gamma)cat$ is defined using a type-definition construct of HOL, where α is the type of pre-objects and $(\alpha \times \gamma \times \alpha)$ is the type of pre-arrows.

```
cat_TY_DEF ⊢def  ∃rep. TYPE_DEFINITION isCat rep
```

B Signature

B.1 Signature as a HOL type

Function names in Ω can be treated as primitives. Functions

$$rho : operator \times sort\ list \times sort \to operator$$
$$arg : operator \times sort\ list \times sort \to sort\ list$$
$$ret : operator \times sort\ list \times sort \to sort$$

obtain the function symbol, the argument type, and the return type of a function name:

```
rho ⊢def  ∀op v s. rho (op,v,s) = op
arg ⊢def  ∀op v s. arg (op,v,s) = v
ret ⊢def  ∀op v s. ret (op,v,s) = s
```

Because we use sets extensively, a predicate *inSet* is defined to test the membership of a set.

```
inSet ⊢def  ∀s e. inSet s e = e IN s
```

A pair (S, Ω) represents a signature if the input and output types of all function-names in Ω are restricted to the sorts in S. Predicate

$$rhoVSRes : sort\ set \rightarrow operator \times sort\ list \times sort \rightarrow bool$$

tests if the input and output types of a function-name (op, sl, s) are restricted to the sorts in S. Predicate $isSig : sort\ set \times (operator \times sort\ list \times sort)set \rightarrow bool$ defines the subset of pairs that are valid representations of signatures.

```
rhoVSRes
⊢def  ∀Ss op sl s. rhoVSRes Ss (op,sl,s) = EVERY (inSet Ss) sl ∧ s IN Ss
isSig
⊢def  ∀Ss Omega. isSig (Ss,Omega) = (∀r ::(inSet Omega). rhoVSRes Ss r)
```

We define a new type *sig* for signatures using HOL's type-definition construct. Accessor functions are defined in HOL to pick out the set of sorts S and the set of function-names Ω of a signature. Tester functions are defined to check, given a signature, whether a sort is in the set of sorts and whether a function-name is in the set of function-names Ω:

```
sig_TY_DEF ⊢def  ∃rep. TYPE_DEFINITION isSig rep
sortsSig ⊢def  ∀x. sortsSig x = FST (REP_sig x)
omegaSig ⊢def  ∀x. omegaSig x = SND (REP_sig x)
inSorts ⊢def  ∀x. inSorts x = inSet (sortsSig x)
inOmega ⊢def  ∀x. inOmega x = inSet (omegaSig x)
```

B.2 Signature as a Category

In HOL, an arrow in category **Sig** is represented as a triple, and the second element as a pair:

$$\begin{aligned}
&(d, (f_s, f_o), c) : \\
&\qquad sig \times \\
&\qquad ((sort \rightarrow sort) \times \\
&\qquad\quad ((operator \times sort\ list \times sort) \rightarrow (operator \times sort\ list \times sort))) \times \\
&\qquad sig
\end{aligned}$$

where $d = (S_1, \Omega_1)$ and $c = (S_2, \Omega_2)$ are the domain and codomain (signatures) of the arrow, and (f_s, f_o) is the signature morphism. Function $f_s : S_1 \rightarrow S_2$ is the sort-mapping function, and $f_o : \Omega_1 \rightarrow \Omega_2$ is the function-name–mapping function.

Two accessor functions are defined to retrieve the sort-mapping and function-name–mapping from an arrow:

```
sigMFs_DEF ⊢def  ∀d fs fo c. sigMFs (d,(fs,fo),c) = fs
sigMFo_DEF ⊢def  ∀d fs fo c. sigMFo (d,(fs,fo),c) = fo
```

Predicate $sigA$ in HOL identifies the signature morphisms from pre-arrow triples $(d, (fs, fo), c)$.

```
sigA_DEF
⊢def  sigA =
   (let SA m =
        isRestr (inSorts (dom m)) (sigMFs m) ∧
        isRestr (inOmega (dom m)) (sigMFo m) ∧
        (∀sn ::(inSorts (dom m)). inSorts (cod m) (sigMFs m sn)) ∧
        (∀r1 ::(inOmega (dom m)).
          let (op,v,s) = r1 and r2 = sigMFo m r1
          in
          (r2 = (rho r2,MAP (sigMFs m) v,sigMFs m s)) ∧
          inOmega (cod m) r2)
    in
    SA)
```

The identity arrow for a signature (S, Ω) is a pair of identity functions $id_S : S \rightarrow S$ and $id_\Omega : \Omega \rightarrow \Omega$. In HOL it is defined as a predicate *sigId*, as follows.

```
sigId
⊢def  sigId = (λs.
   (s, ((λs ::(inSorts s). s), (λr ::(inOmega s). r)), s))
```

Composition of two arrows m and n is defined as the pairwise composition of their components. The HOL definition of composition operation *sigOo* is as follows.

```
sigOo
⊢def  sigOo =
   (λm.
     λn ::(cpsl sigA m).
       dom n,
       ((λx ::(inSorts (dom n)). (sigMFs m o sigMFs n) x),
        (λx ::(inOmega (dom n)). (sigMFo m o sigMFo n) x)),
       cod m)
```

We have proved that the so-defined four-tuple $(\lambda s.T, sigA, sigId, sigOo)$ is indeed a category called *sigCat*.

```
sigCat_rep_IS_CAT
[oracles: #] [axioms: ] [] ⊢  isCat ((λs. T), sigA, sigId, sigOo)
sigCat ⊢def  sigCat = ABS_cat ((λs. T), sigA, sigId, sigOo)
```

C Σ-Algebra

C.1 Algebra as a HOL type

Triple (Σ, A, I) represents a Σ-algebra in HOL, where A is a HOL function that constructs a carrier set, given a sort of the signature Σ, and I is a HOL function that maps a function-name to a function:

$$A : sort \rightarrow value\ set$$
$$I : (operator \times sort\ list \times sort) \rightarrow (value\ list \rightarrow value)$$

Function A is called the carrier-sets assignment function and I the function-name assignment function.

The HOL function $carrierVals$ computes the carrier set of a Σ-algebra, given the carrier-sets assignment function A.

```
carrierVals
⊢def  ∀A Ss. carrierVals A Ss = {(s, v) | inSet Ss s ∧ inSet (A s) v}
```

The Function $setOfVLists$ constructs from the carrier-sets assignment function A and a sort list sl a set, such that every element of the set is a $value$ list, and each $value$ in the list has the type defined by the corresponding element in the sort list sl:

```
setOfVLists
⊢def  ∀A v. setOfVLists A v = {xv | AND_EL (MAP2 inSet (MAP A v) xv)}
```

A pre-algebra triple (Σ, A, I) is a Σ-algebra if the mapping from function-names to functions is consistent with the mapping from sorts to carrier sets. In HOL, this property is defined as $isAlg$:

```
isAlg
⊢def  ∀sigma A Is.
     isAlg (sigma,A,Is) =
     isRestr (inSorts sigma) A ∧
     isRestr (inOmega sigma) Is ∧
     (∀rvs ::(inOmega sigma).
       isRestr (inSet (setOfVLists A (arg rvs))) (Is rvs) ∧
       (∀xv ::(inSet (setOfVLists A (arg rvs))).
         inSet (A (ret rvs)) (Is rvs xv)))
```

Predicate $isAlg$ defines the subset of triples that are valid representations of type alg for algebras. We define type alg for algebras using HOL's type-definition construct.

```
alg_TY_DEF ⊢def  ∃rep. TYPE_DEFINITION isAlg rep
alg_ISO_DEF
⊢def  (∀a. ABS_alg (REP_alg a) = a) ∧
   (∀r. isAlg r = REP_alg (ABS_alg r) = r)
```

Accessor functions $sigAlg, carrierAlg$, and $funsAlg$ extract the signature, the carrier sets, and the interpretation of function names from an algebra.

```
sigAlg ⊢def  ∀x. sigAlg x = FST (REP_alg x)
carrierAlg ⊢def  ∀x. carrierAlg x = FST (SND (REP_alg x))
funsAlg ⊢def  ∀x. funsAlg x = SND (SND (REP_alg x))
```

C.2 Σ-Algebra as a Category

Category $\mathbf{Alg}_\Sigma$ is a category of Σ-algebras whose objects are Σ-algebras and whose arrows are Σ-homomorphisms. A Σ-homomorphism between (A, I) and

(A', I') is a function $h : A \to A'$ such that

$$h(I_\rho(v_1, v_2, ..., v_n)) = I'_\rho(h\ v_1, h\ v_2, \ldots, h\ v_n)$$

Each signature Σ uniquely defines a category $\mathbf{Alg}_\Sigma$. We have defined category $\mathbf{Alg}_\Sigma$ in such way that the signature Σ is taken as a parameter. Predicate *isSigmaAlg* selects Σ-algebras from the set of all algebras.

```
isSigmaAlg
⊢def  ∀sigma. isSigmaAlg sigma = (let P a = (sigAlg a = sigma) in P)
```

In HOL, an arrow in category $\mathbf{Alg}_\Sigma$ is represented as a triple:

$$(m, h, n) : alg \times (sort \to value \to value) \times alg$$

where $m = (\Sigma, A, I)$ and $n = (\Sigma', A', I')$ are domain and codomain Σ-algebras of the arrow, and $h : sort \to value \to value$ is a function that maps elements in carrier-sets of m to carrier-sets of n. The predicate *algHom* defines a homomorphism between two algebras that have the same signature. The predicate *sigmaAlgA* picks out $\mathbf{Alg}_\Sigma$ arrow from pre-arrows with the help of *algHom*.

```
algHom_DEF
⊢def  algHom =
   (let aA (m,h,n) =
        (let values =
             carrierVals (carrierAlg m) (sortsSig (sigAlg m))
         in
         (sigAlg m = sigAlg n) ∧
         isRestr (inSet (sortsSig (sigAlg m))) h ∧
         (∀s ::(inSet (sortsSig (sigAlg m))).
           isRestr (inSet (carrierAlg m s)) (h s) ∧
           (∀rvs ::(inOmega (sigAlg m)).
             ∀xv ::(inSet (setOfVLists (carrierAlg m) (arg rvs))).
               h (ret rvs) (funsAlg m rvs xv) =
               funsAlg n rvs (MAP2 h (arg rvs) xv))))
    in
    aA)

sigmaAlgA
⊢def  ∀sigma.
      sigmaAlgA sigma =
      (let AA (m,h,n) = isSigmaAlg sigma m ∧ algHom (m,h,n) in AA)
```

The identity arrow in category $\mathbf{Alg}_\Sigma$ is the function $(\lambda s.I : value \to value)$. The composition of arrows (a, f, b) and (b, g, c) is the composition of functions $(a, \lambda s.(g\ s) \circ (f\ s), c)$.

The four-tuple *algCat* that represents the $\mathbf{Alg}_\Sigma$-category is defined as follows.

```
sigmaAlgO
⊢def  ∀sigma. sigmaAlgO sigma = (let AO = isSigmaAlg sigma in AO)
sigmaAlgId
⊢def  ∀sigma.
     sigmaAlgId sigma =
     (λa ::(sigmaAlgO sigma). a, (λs ::(inSorts sigma). I), a)
sigmaAlgOo
⊢def  ∀sigma.
     sigmaAlgOo sigma =
     (λm.
       λn ::(cpsl (sigmaAlgA sigma) m).
         dom n, (λs ::(inSorts sigma). mid m s o mid n s), cod m)
algCat
⊢def  ∀sigma.
     algCat sigma =
     (sigmaAlgO sigma,
      sigmaAlgA sigma,
      sigmaAlgId sigma,
      sigmaAlgOo sigma)
```

To prove that *algCat* represents a category, we need to prove the following goal.

```
val goal = ([],  --`isCat (algCat sigma)`--);
```

C.3 Terms of Σ-Algebra

A variable assignment assigns types to free variable symbols and must be consistent with the set of sorts, for it to be a valid assignment. Property *gammaRes* states that the variable assignment Γ is restricted to the set of sorts S.

```
gammaRes
⊢def  ∀sigma gamma.
     gammaRes sigma gamma =
     (∀(v,s) ::(inSet gamma). inSorts sigma s)
```

We first define a HOL-type *ppreT* as a pre-$\Sigma_{pre}(\Gamma)$ term. Pre-$\Sigma_{pre}(\Gamma)$ terms are represented as recursive-type *ppreT* defined in HOL using a recursive-type definition package. A pre-$\Sigma_{pre}(\Gamma)$ term contains the type information of the term.

```
ppreT ::= Leafv (v, s) |          (a variable)
          Leafo (op, sl, s) |     (a function symbol)
          Comb ppreT ppreT;
```

In HOL, this syntax is represented by recursive-type *ppreT* with three constructors:

$$Leafv : (variable \times sort) \rightarrow ppreT$$
$$Leafo : (operator \times sort\ list \times sort) \rightarrow ppreT$$
$$Comb : ppreT \rightarrow ppreT \rightarrow ppreT$$

We can now define $\Sigma_{pre}(\Gamma)$ term. A term of $\Sigma_{pre}(\Gamma)$ has type $(sort\ list \times sort)$ and is represented as a binary tree. The terms of $\Sigma_{pre}(\Gamma)$ are defined inductively:

$$\frac{}{isPreSigmaTerm\ sigma\ gamma\ ([\,], s)(Leafv\ (x, s))}$$

$$\frac{}{isPreSigmaTerm\ sigma\ gamma\ (sl, s)(Leafo\ (op, sl, s))}$$

$$\frac{\begin{array}{c} isPreSigmaTerm\ sigma\ gamma\ (v_1, s_1)t_1, \\ isPreSigmaTerm\ sigma\ gamma\ ([\,], s_2)t_2 \end{array}}{isPreSigmaTerm\ sigma\ gamma\ (TL\ v_1, s_1)(Comb\ t_1\ t_2)}\ s_2 = HD\ v_1$$

A $\Sigma(\Gamma)$-term of type s is just a $\Sigma_{pre}(\Gamma)$ term whose type is $([\,], s)$. $\Sigma(\Gamma)$-terms are represented as four-tuples $(\Sigma, \Gamma, t, s) : sig \times (variable \times sort)set \times ppreT \times sort$. Predicate $isSigmaTerm$ defines the subset of the four-tuples that are valid representations of HOL-type $sigmaTm$ for Σ-terms.

```
isSigmaTerm
⊢def  ∀sigma gamma t s.
        isSigmaTerm (sigma,gamma,t,s) =
        isPreSigmaTerm sigma gamma ([],s) t
```

We then define Σ-terms as a HOL-type $sigmaTm$ through HOL's type-definition construct:

```
sigmaTm_TY_DEF
[oracles: #] [axioms: ] [] ⊢def  ∃rep. TYPE_DEFINITION isSigmaTerm rep
sigmaTm_ISO_DEF
[oracles: #] [axioms: ] []
⊢def  (∀a. ABS_sigmaTm (REP_sigmaTm a) = a) ∧
    (∀r. isSigmaTerm r = REP_sigmaTm (ABS_sigmaTm r) = r)
```

Accessor functions $sigSigmaTm, gammaSigmaTm, tmSigmaTm$, and $tySigmaTm$ extract the signature, the variable assignment, the term itself, and its type from a Σ-term.

```
sigSigmaTm ⊢def  ∀x. sigSigmaTm x = FST (REP_sigmaTm x)
gammaSigmaTm ⊢def  ∀x. gammaSigmaTm x = FST (SND (REP_sigmaTm x))
tmSigmaTm ⊢def  ∀x. tmSigmaTm x = FST (SND (SND (REP_sigmaTm x)))
tySigmaTm ⊢def  ∀x. tySigmaTm x = SND (SND (SND (REP_sigmaTm x)))
```

A Σ-equation is a pair of terms $(t_1, t_2) : sigmaTm \times sigmaTm$ where t_1 and t_2 are of the same type. Predicate $isSigmaEq$ selects Σ-equations from pre-equation pairs. It defines the subset of pre-equations that are valid representation of HOL-type $sigmaEq$ for Σ-equations.

```
isSigmaEq
⊢def  ∀t1 t2.
        isSigmaEq (t1,t2) =
        (sigSigmaTm t1 = sigSigmaTm t2) ∧
        (gammaSigmaTm t1 = gammaSigmaTm t2) ∧
        (tySigmaTm t1 = tySigmaTm t2)
```

We then define Σ-equations as type *sigmaEq* through HOL's type definition construct:

```
sigmaEq_TY_DEF ⊢def  ∃rep. TYPE_DEFINITION isSigmaEq rep
sigmaEq_ISO_DEF
⊢def  (∀a. ABS_sigmaEq (REP_sigmaEq a) = a) ∧
   (∀r. isSigmaEq r = REP_sigmaEq (ABS_sigmaEq r) = r)
```

Accessor functions $leftTermEq$ and $rightTermEq$ extract the left and right terms from a Σ-equation.

```
leftTermEq ⊢def  ∀x. leftTermEq x = FST (REP_sigmaEq x)
rigthTermEq ⊢def  ∀x. rightTermEq x = SND (REP_sigmaEq x)
```

Functions $sigSigmaEq, gammaSigmaEq, ltmSigmaEq, rtmSigmaEq$, and $tySigmaEq$ obtain the signature, the variable assignment, the left term itself, the right term itself, and the type of the terms.

```
sigSigmaEq ⊢def  ∀x. sigSigmaEq x = sigSigmaTm (leftTermEq x)
gammaSigmaEq ⊢def  ∀x. gammaSigmaEq x = gammaSigmaTm (leftTermEq x)
ltmSigmaEq ⊢def  ∀x. ltmSigmaEq x = tmSigmaTm (leftTermEq x)
rtmSigmaEq ⊢def  ∀x. rtmSigmaEq x = tmSigmaTm (rightTermEq x)
tySigmaEq ⊢def  ∀x. tySigmaEq x = tySigmaTm (leftTermEq x)
```

D Algebraic Specifications

D.1 Specification as a HOL type

In HOL, specifications are represented by a triple $(\Sigma, \Gamma, E) : sig \times (variable \times sort)\ set \times sigmaEq\ set$ where Σ is a signature, Γ is a variable assignment, and E is a set of Σ-equations.

Predicate *isSpec* defines the subset of pre-specification triples that are valid representations of the HOL-type *spec* for algebraic specifications.

Predicate *isSpec* picks out specifications from pre-specification triples.

```
isSpec
⊢def  ∀sigma gamma E.
     isSpec (sigma,gamma,E) =
     (∀sigEq ::(inSet E).
       (sigSigmaEq sigEq = sigma) ∧ (gammaSigmaEq sigEq = gamma))
```

We then define algebraic specifications as a type *spec* through HOL's type-definition construct:

```
spec_TY_DEF ⊢def  ∃rep. TYPE_DEFINITION isSpec rep
spec_ISO_DEF
⊢def  (∀a. ABS_spec (REP_spec a) = a) ∧
   (∀r. isSpec r = REP_spec (ABS_spec r) = r)
```

Accessor functions $sigSpec, gammaSpec$, and $ESpec$ extract the signature, the variable assignment, and the set of Σ-equations from a specification.

```
sigSpec ⊢def   ∀x. sigSpec x = FST (REP_spec x)
gammaSpec ⊢def   ∀x. gammaSpec x = FST (SND (REP_spec x))
ESpec ⊢def   ∀x. ESpec x = SND (SND (REP_spec x))
```

D.2 Algebraic Specifications as a Category

A specification arrow is represented as a triple:

$$
\begin{aligned}
&(d, (fs, fo, fv), c) : \\
&\mathit{spec} \times \\
&((\mathit{sort} \rightarrow \mathit{sort}) \times \\
&\quad ((\mathit{operator} \times \mathit{sort\ list} \times \mathit{sort}) \rightarrow (\mathit{operator} \times \mathit{sort\ list} \times \mathit{sort})) \\
&\quad ((\mathit{variable} \times \mathit{sort}) \rightarrow (\mathit{variable} \times \mathit{sort}))) \times \\
&\mathit{spec}
\end{aligned}
$$

Accessor functions are defined to retrieve sort-mapping, function-name–mapping, and variable mapping from a specification arrow:

```
specMFs_DEF ⊢def   ∀d fs fo fv c. specMFs (d,(fs,fo,fv),c) = fs
specMFo_DEF ⊢def   ∀d fs fo fv c. specMFo (d,(fs,fo,fv),c) = fo
specMFv_DEF ⊢def   ∀d fs fo fv c. specMFv (d,(fs,fo,fv),c) = fv
```

The signature arrow $(f_s, f_o) : (S, \Omega) \rightarrow (S', \Omega')$ and variables mapping $f_v : \Gamma \rightarrow \Gamma'$ decide how to transform the Σ-equations of one specification to the Σ-equations of another specification. First, terms of pre-$\Sigma_{pre}(\Gamma)$ are transformed to that of pre-$\Sigma'_{pre}(\Gamma')$ by HOL function $transPPT$.

```
transPPT
[oracles: #] [axioms: ] []
⊢def   (∀fo fv x. transPPT fo fv (Leafv x) = Leafv (fv x)) ∧
    (∀fo fv op. transPPT fo fv (Leafo op) = Leafo (fo op)) ∧
    (∀fo fv ppt2 ppt1.
      transPPT fo fv (Comb ppt1 ppt2) =
      Comb (transPPT fo fv ppt1) (transPPT fo fv ppt2))
```

Second, $\Sigma(\Gamma)$-terms t of type s is transformed to term $t' = tranPPT\ t$ of type $(f_s\ s)$:

```
transTerm
⊢def   ∀fs fo fv sigma2 gamma2 t.
      transTerm fs fo fv sigma2 gamma2 t =
      ABS_sigmaTm
        (sigma2,gamma2,transPPT fo fv (tmSigmaTm t),fs (tySigmaTm t))
```

Last, Σ-equation $t_1 =_s t_2$ is transformed to $t_1' =_{s'} t_2'$, where $t_1' = \mathit{transPPT}\ t_1$, $t_2' = \mathit{transPPT}\ t_2$ and $s' = (f_s\ s)$:

```
transEq
⊢def  ∀fs fo fv sigma2 gamma2 eq.
     transEq fs fo fv sigma2 gamma2 eq =
     ABS_sigmaEq
       (ABS_sigmaTm
          (sigma2,
           gamma2,
           transPPT fo fv (ltmSigmaEq eq),
           fs (tySigmaEq eq)),
        ABS_sigmaTm
          (sigma2,
           gamma2,
           transPPT fo fv (rtmSigmaEq eq),
           fs (tySigmaEq eq)))
```

We use a constant function $thmsDerivable : sigmaEq\ set \rightarrow sigmaEq\ set$ that forms a theory, given a set of axioms.

Now we can define a specification arrow in HOL. Predicate *inGammaSpec* defines the elements that are in the variable-assignment set Γ of a specification. Predicate *specA* picks out the specification arrows from pre-arrow triples.

```
inGammaSpec ⊢def  ∀sp. inGammaSpec sp = inSet (gammaSpec sp)
specA_DEF
⊢def  specA =
   (let SA (d,(fs,fo,fv),c) =
        sigA (sigSpec d,(fs,fo),sigSpec c) ∧
        isRestr (inGammaSpec d) fv ∧
        (∀(x1,s1) ::(inGammaSpec d).
          let (x2,s2) = fv (x1,s1)
          in
          inGammaSpec c (x2,s2) ∧ (s2 = fs s1)) ∧
        (∀eq ::(inSet (ESpec d)).
          inSet (thmsDerivable (ESpec c))
            (transEq fs fo fv (sigSpec c) (gammaSpec c) eq))
    in
    SA)
```

We define function *sigASpecA* to extract the signature arrow from a specification arrow.

```
sigASpecA_DEF
⊢def  sigASpecA =
   (let sAsA (d,(fs,fo,fv),c) = sigSpec d,(fs,fo),sigSpec c in sAsA)
```

The identity arrow of object (Σ, E) in category **Spec** is that of object Σ in category **Sig**.

```
specId_DEF
⊢def  ∀sp.
      specId sp =
      (let (fs,fo) = mid (sigId (sigSpec sp))
       and fv = (λx ::(inGammaSpec sp). x)
       in
       sp,(fs,fo,fv),sp)
```

The composite arrow in category **Spec** is defined the same way as in category **Sig**.

```
specOo_DEF
⊢def  specOo =
   (λm.
     λn ::(cpsl specA m).
       let (fs,fo) = mid (sigOo (sigASpecA m) (sigASpecA n))
       and fv =
           (λx ::(inGammaSpec (dom n)). (specMFv m o specMFv n) x)
       in
       dom n,(fs,fo,fv),cod m)
specCat_REP ⊢def  specCat = ((λx. T),specA,specId,specOo)
```

To prove that *specCat* represents a category, we need to prove the following goal.

```
val goal = ([],  --`isCat specCat`--);
```