AFRL-IF-RS-TR-2000-18
Final Technical Report
March 2000

PRAGMATIC APPROACHES TO COMPOSITION AND VERIFICATION OF ASSURED SOFTWARE

Syracuse University

Dan Zhou, Susan Older, and Shiu-Kai Chin

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2000-18 has been reviewed and is approved for publication.

APPROVED: Roy F. Stratton

ROY F. STRATTON
Project Engineer

FOR THE DIRECTOR: Northrup Fowler

NORTHRUP FOWLER
Technical Advisor
Information Technology Division

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	MARCH 2000	Final Feb 98 - Aug 99

4. TITLE AND SUBTITLE
PRAGMATIC APPROACHES TO COMPOSITION AND VERIFICATION OF ASSURED SOFTWARE

5. FUNDING NUMBERS
C - F30602-98-1-0063
PE - 61102F
PR - 2304
TA - FR
WU - P8

6. AUTHOR(S)
Dan Zhou, Susan Older, and Shiu-Kai Chin

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
CASE Center
Syracuse University
2-212 Center for Science and Technology
Syracuse NY 13244-4100

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/IFTD
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2000-18

11. SUPPLEMENTARY NOTES
Air Force Research Laboratory Project Engineer: Roy F. Stratton/IFTD/(315) 330-3004
Initiating Project Engineer: Major Mark J. Gerken
This effort was funded by Air Force Office of Scientific Research

12a. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Mature engineering fields have methods of construction that have high likelihoods of success, and that guarantee the proper functioning of systems, even within hostile environments. These methods relate behavior to structure and have underlying notions of composition related to the implementation domain. Unfortunately, the construction of computer systems has not yet reached the same level of maturity. While many mathematical theories have been developed, they have not yet been brought into standard engineering practice.
Bridging the gap between theory and engineering practice requires sound and pragmatic principles of construction and composition for software systems. One potentially promising and practical approach employs a combination of higher-order logic, category theory, and algebraic specifications, as incorporated into the HOL theorem prover and the Specware system for specification composition, refinement, and code synthesis.
This report presents a HOL formulation of the primary mathematical concepts underlying Specware, fully explicating the underlying principles of construction and composition. Furthermore, the purpose of computer-assisted reasoning is to allow nonexperts in a given domain to nonetheless have confidence in their analysis. The HOL formulation describes the relevant concepts in an executable form that nonexperts can use in the future to construct assured specifications and ultimately assured code.

14. SUBJECT TERMS
Formal Methods, Higher Order Logic, Software, Algebraic Specifications, Verification, Assured Software, Theorem Prover

15. NUMBER OF PAGES
32

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

1 Introduction

Mature engineering fields have methods of construction that have a high likelihood of success and that guarantee the proper functioning of systems, even within hostile environments. These methods relate behavior to structure and have some underlying notion of composition related to the implementation domain. Unfortunately, the construction of computer systems has not yet reached the same level of maturity. While many mathematical theories have been developed, they have not yet been brought into standard engineering practice.

Bridging this gap between theory and engineering practice requires sound and pragmatic principles of construction and composition for software systems. Thus there are at least two necessary tasks: *identifying* these principles, and *investigating* their suitability for problems of real engineering interest. Our approach is to adopt existing theories and technology where possible and to explore how they can be applied to nontrivial engineering applications. In particular, we focus on higher-order logic, category theory, and algebraic specifications, making significant use of the *Higher Order Logic* (HOL) theorem-prover [2] and Kestrel Institute's SPECWARE specification composition and refinement system [3].

The method of design in both HOL and SPECWARE is to construct small modules that can be composed and verified. The well-documented advantages of modularity apply here, as modular theories will be more reusable, and easier to build and verify. HOL theories are organized hierarchically, so that new theories can be built by specialization of existing theories. Design in SPECWARE is a semi-automatic process, in which the designer creates specifications and chooses composition or refinement methods, which are performed automatically by the system. Again, the creation of small specifications is the preferred method. The universal composition method, based on pushouts and colimits in category theory, composes specifications in a canonical way. The refinement methods can create either C++ or LISP code.

Our overall approach is to build HOL theories that specify the desirable properties and invariants that characterize the task, and use HOL's theorem-proving capability to verify the soundness and completeness of the collection of theories. These theories are then transformed into SPECWARE specifications, which are then refined into executable code. This approach has been used to formally define and specified much of a secure electronic mail protocol, RFC 1421 – Privacy Enhanced Mail, [4]; these results have been reported elsewhere [9, 10, 8].

HOL theories and SPECWARE specifications are both higher-order theories, so the mapping between them is fairly straightforward. However, there is a technical difficulty in the refinement process, because there are many potential refinements of a SPECWARE specification. Furthermore, not all refinements result in consistent specifications (i.e., specifications which can be refined to meaningful and valid code). Ultimately, we would like to identify explicit principles of construction that ensure the appropriate refinements and to explore the applicability of these principles.

This report describes an important first step, namely the formulation in higher-order logic of the primary concepts that underlie SPECWARE's refinement framework. Throughout this report, we provide both high-level, English-language explanations of the concepts, followed by their implementation in the logic of the HOL theorem prover. Section 2 covers the most basic definitions of *category theory* [6], the primary foundation for the rest of the mathematical framework. The next three sections describe the foundations of *algebraic specifications* [1]. Section 3 introduces *signatures*, which are (roughly speaking) high-level abstractions that identify the basic data types and the basic operators of a system. *Algebras*—which provide interpretations for these signatures—appear in Section 4. To constrain the possible interpretations of a signature, it is necessary to introduce further constraints, leading to *specifications*; these are discussed in Section 5. Finally, Section 6 describes possible future work that carries our approach further.

2 Category-Theory Basics

In this section, we give an overview of the most basic definitions of category theory (including the notion of *category* itself) necessary for understanding the mathematical framework underlying SPECWARE. Our aim here is to provide English-language explanations of the concepts as well as their formulations in the higher-order logic of the HOL theorem prover [2]. In fact, we shall follow this approach throughout this report.

The definitions in this section are not original, either in their English forms or in their HOL forms. For example, Pierce provides a significantly more complete introduction to category theory [6]. The HOL formulations we give in this section are due to Morris [5] and will form the basis of our own formulations in subsequent sections; Agerholm provides a similar embedding of category theory into HOL, choosing a different representation for the categorical arrows [7].

A *category* $\mathcal{C}$ comprises a collection $O_{\mathcal{C}}$ of *objects* and a collection $A_{\mathcal{C}}$ of *arrows* satisfying the properties detailed below. We often refer to the elements of $O_{\mathcal{C}}$ as $\mathcal{C}$-objects and to the elements of $A_{\mathcal{C}}$ as $\mathcal{C}$-arrows.

- Each arrow is associated with two objects called its *domain* and *codomain*. When f is an arrow whose domain and codomain are A and B, respectively, we write $f : A \rightarrow B$.
- For each $\mathcal{C}$-object A, there is an identity arrow $id_A : A \rightarrow A$.
- For each pair of $\mathcal{C}$-arrows $f : A \rightarrow B$ and $g : B \rightarrow C$, there is a composite arrow $g \circ f : A \rightarrow C$. The composition operator $\circ$ satisfies the following properties:

  Identity For any $\mathcal{C}$-arrow $f : A \rightarrow B$,

  $$f \circ id_A = f \qquad \text{and} \qquad id_B \circ f = f$$

  Associativity For any $\mathcal{C}$-arrows $f : A \rightarrow B$, $g : B \rightarrow C$, and $h : C \rightarrow D$,

  $$h \circ (g \circ f) = (h \circ g) \circ f$$

For example, the category **Set** is a category of sets (as objects) and total functions between sets (as arrows). The identity arrow is the identity function, and the composition in the category is the standard function composition.

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

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

satisfying the following constraints:

- The function O picks out C-objects from pre-objects of HOL type α.
- The function A picks out C-arrows from pre-arrows of type $\alpha \times \gamma \times \alpha$.
- The function Id constructs an identity arrow for each $\mathcal{C}$-object.
- The function oo constructs a composite arrow for two $\mathcal{C}$-arrows f and g, provided that they are composable (i.e., when the domain of g is the codomain of f).

As a technical aside, we point out the name oo is used for the categorical composition operator to avoid confusion with HOL's built-in composition operator **o**.

Each arrow in the category is represented by a triple (d, f, c) of type $\alpha \times \gamma \times \alpha$, where d, f and c correspond to the arrow's domain, the arrow itself, and its codomain, 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 the property *cpsl* asserts that two triples are arrows of a certain category and that they are composable. These properties are summarized as follows:

```
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 the value ARB. The 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 *isRestr*2 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)
```

Using these definitions, the predicate *isCat* (given in Figure 1) checks whether a given four-tuple is indeed a category. It is straightforward to see that this HOL formulation captures all of the important aspects of the definition of *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)))
```

Figure 1: The predicate `isCat`.

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
```

3 Signatures

Having provided a HOL formulation of categories in the last section, we now turn our attention to some particular categories of importance to the development of assured code via algebraic specifications. In this section, we consider *signatures*, which introduce a collection of data types and operations on those types. Signatures are purely syntactic entities and have no meanings in and of themselves; in Section 4, we will examine *algebras*, which provide meanings for these signatures.

A signature Σ is a pair (S,Ω), where S is a set of *sorts* (intuitively, base types) and Ω is a set of function symbols (also called *operators*). Each function symbol ρ in Ω has an associated type $(s_1 \times s_2 \times s_3 \times \cdots \times s_n) \rightarrow s_0$ for some $n \geq 0$, with each s_i (for $i \in \{0,1,\cdots,n\}$) a member of S; such an operator is said to have arity n. A function symbol with type $\rightarrow s$ is called a constant and is said to have type s.

$$
\begin{array}{l}
\Sigma_{tl}: \\
\quad S_{tl}\ (sorts): \\
\qquad color \\
\quad \Omega_{tl}\ (function\ symbols): \\
\qquad green : color \\
\qquad yellow : color \\
\qquad red : color \\
\qquad changeColor : color \rightarrow color
\end{array}
\qquad
\begin{array}{l}
\Sigma_{bp}: \\
\quad S_{bp}\ (sorts): \\
\qquad boolPair \\
\quad \Omega_{bp}\ (function\ symbols): \\
\qquad TT : boolPair \\
\qquad TF : boolPair \\
\qquad FT : boolPair \\
\qquad FF : boolPair \\
\qquad cycle : boolPair \rightarrow boolPair
\end{array}
$$

Figure 2: Sample signatures.

For example, Figure 2 contains two signatures, Σ_{tl} (for traffic lights) and Σ_{bp} (for boolean pairs). The traffic-light signature Σ_{tl} has a sort *color* to represent the colors of a traffic light, and three operators (i.e., *green*, *red*, and *yellow*) of type *color* to represent the three possible colors of a traffic light. It also has an operator *changeColor* that changes the color of the traffic light. (To be pedantic here, our *intention* is that *changeColor* will eventually have a certain behavior. However, because we are currently at a purely syntactic level, we are only indicating that there will be *some* operator with this name.)

Likewise, the boolean-pair has a sort *boolPair* to represent boolean pairs and four operators (i.e., TT, TF, FT, FF) of type *boolPair* intended to represent the four possible combinations for a boolean pair. It also has an operator *cycle* intended to cycle through the various boolean pairs in a particular sequence.

3.1 Signatures 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 with special properties.

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 $(\rho, sl, s) : operator \times sort\ list \times sort$, 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 with type s is represented by a triple $(c, [\,], s)$, where $[\,]$ is HOL's representation of the empty list. Using triples to represent the elements of the set Ω allows us to overload functions symbols. Elements (which we shall call *function names*) in Ω can be treated as primitives. We define accent Functions

$$rho : operator \times sort\ list \times sort \rightarrow operator,$$
$$arg : operator \times sort\ list \times sort \rightarrow sort\ list,$$
$$ret : operator \times sort\ list \times sort \rightarrow sort$$

to obtain the function symbol, the argument type, and the return type of a given 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. The predicate

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

tests whether the input and output types of a function-name (op, sl, s) are restricted to the sorts in S. The predicate $isSig : sort\ set \times (operator \times sort\ list \times sort)set \rightarrow bool$ then 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)
```

Finally, we define a new type *sig* for signatures using HOL's type-definition construct. Accessor functions are defined in HOL to pick out from a signature its set of sorts S and its set of function names Ω. We also define tester functions that, given a signature, check 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)
```

3.2 Signatures as a Category

A signature morphism f between two signatures $\Sigma = (S, \Omega)$ and $\Sigma' = (S', \Omega')$ is a pair of functions $f_s : S \to S'$ and $f_o : \Omega \to \Omega'$ such that the mapping f_o between function symbols respects the mapping f_s between sorts. That is, if a function-symbol $\rho \in \Omega$ has type $(s_1 \times s_2 \times s_3 \times \cdots \times s_n) \to s_0$, then $f_o(\rho)$ is a function symbol of Ω' having type $(f_s(s_1) \times f_s(s_2) \times \cdots \times f_s(s_n)) \to f_s(s_0)$.

For example, recall the signatures of Figure 2. If the sort mapping $f_s : S_{tl} \to S_{bp}$ maps the traffic light's *color* sort to the boolean pair's *boolPair* sort, then $f_o : \Omega_{tl} \to \Omega_{bp}$ should map *green* to a function symbol in Ω_{bp} that has type *boolPair*. Thus f_o can map *green* to any of the four operators of type *boolPair* (TT, TF, FT, FF) in Ω_{bp}, but it cannot map *green* to *cycle*.

We can now define the category **Sig** whose objects are signatures and whose arrows are signature morphisms. In HOL, we represent an arrow in the category **Sig** by a triple $(d, (f_s, f_o), c)$ having the following type:

$$\begin{aligned}&\mathit{sig} \times \\ &((\mathit{sort} \to \mathit{sort}) \times \\ &\quad ((\mathit{operator} \times \mathit{sort\ list} \times \mathit{sort}) \to (\mathit{operator} \times \mathit{sort\ list} \times \mathit{sort}))) \times \\ &\mathit{sig}\end{aligned}$$

In this representation, d and c are the signatures that serve as domain and codomain of the arrow, and (f_s, f_o) is the signature morphism itself.

We first define two accessor functions that retrieve the sort-mapping and operator-mapping components of 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
```

We then define a predicate *sigA*, which identifies the signature morphisms from pre-arrow triples $(d, (fs, fo), c)$. In particular, it ensures that fs and fo are indeed functions from the sorts and operators of d to the sorts and operators of c; it also checks that the funciton-name mapping fo respects the sort mapping fs.

```
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 $id_S : S \to S$ and $id_\Omega : \Omega \to \Omega$. 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))
```

The composition of two signature morphisms m and n is defined compentwise, so that their sort-mapping functions are composed and their operator-mapping functions are composed. The HOL definition of this 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)
```

These definitions together allow us to prove that the four-tuple $(\lambda s.T, sigA, sigId, sigOo)$ is indeed a category (which we call *sigCat*), as evidenced by the following HOL theorem.

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

sigCat ⊢def  sigCat = ABS_cat ((λs. T), sigA, sigId, sigOo)
```

4 Algebras

Algebras give meaning to signatures. For any given signature, there are many potential algebras, each providing a different interpretation of the sorts and function symbols. For a fixed signature $\Sigma = (S, \Omega)$, we can talk about its collection of models; these models are called Σ-algebras.

Each Σ-algebra is a pair $(\mathcal{A}, \mathcal{I})$, where $\mathcal{A} = \{A_s \mid s \in S\}$ is an S-indexed set of *carriers*, and $\mathcal{I} = \{I_\rho \mid \rho \in \Omega\}$ is an Ω-indexed set of functions. Furthermore, we require the functions I_ρ to respect the typings of each ρ: if the function symbol ρ is assigned the type $s_1 \times \cdots \times s_n \to s$, then I_ρ must be a function of type $(A_{s_1} \times A_{s_2} \cdots \times A_{s_n}) \to A_s$. Intuitively, each carrier A_s provides a set of values corresponding to the sort s. In turn, each function I_ρ provides an interpretation of the function symbol ρ as a function over the appropriate sets of data values.

For example, recall the boolean-pair signature Σ_{bp} from Figure 2. One possible Σ_{bp}-

algebra is $(\{A_{boolPair}\}, \{I_{TT}, I_{TF}, I_{FT}, I_{FF}, I_{cycle}\})$, where:

$$\begin{aligned}
A_{boolPair} &= \{(T,T),(T,F),(F,T),(F,F)\} \\
I_{TT} &= (T,T) \\
I_{TF} &= (T,F) \\
I_{FT} &= (F,T) \\
I_{FF} &= (F,F) \\
I_{cycle} &= \lambda(x : A_{boolPair}). \\
&\qquad \text{if } x = (F,F) \text{ then } (F,T) \\
&\qquad \text{else if } x = (F,T) \text{ then } (T,F) \\
&\qquad \text{else if } x = (T,F) \text{ then } (T,T) \\
&\qquad \text{else } (F,F)
\end{aligned}$$

Note that while this algebra reflects our probable *intended* interpretation of the operators TT, TF, FT, and FF, this interpretation is *not* the only one. For example, consider the (rather artificial) algebra $(\{B_{boolPair}\}, \{J_{TT}, J_{TF}, J_{FT}, J_{FF}, J_{cycle}\})$, where $B_{boolPair} = \mathbb{N}$ (the set of natural numbers) and the J_ρ functions are defined as follows:

$$\begin{aligned}
J_{TT} &= 0 \\
J_{TF} &= 5 \\
J_{FT} &= 5 \\
J_{FF} &= 13 \\
J_{cycle} &= \lambda(x : \mathbb{N}).\ \text{if } x < 2 \text{ then } 0 \text{ else } 3
\end{aligned}$$

This algebra meets all the necessary requirements: each constant operator of sort *boolPair* is mapped to a value from the set $B_{boolPair}$, and the operator *cycle* is mapped to a function of type $B_{boolPair} \rightarrow B_{boolPair}$. In particular, it is not necessary for the values of $B_{boolPair}$ to resemble boolean pairs or to even be in one-to-one correpsondence with the constants of the sort *boolPair*. Similarly, it is okay for different constants to be mapped to the same value, and the function J_{cycle} can return values that are not necessarily assigned to a particular constant.

In Section 5, we will discuss how signatures can be augmented with additional formulas or equations that rule out certain undesirable algebras (such as this one, perhaps). For now, however, we focus on the HOL implementation of algebras.

4.1 Algebras as a HOL Type

We introduce 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. Similarly, 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 triples of form (Σ, A, I) to represent Σ-algebras in HOL: in each case, A is a HOL function that constructs a carrier set from the sort of the signature Σ, and I is a HOL function that maps a function-name to a function. Thus A and I have the following types in HOL:

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

We call A 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}
```

Likewise, the function *setOfVLists* constructs a set from the carrier-sets assignment function A and a sort list *sl* 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 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 a predicate *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)))
```

This predicate *isAlg* identifies the subset of triples that are valid representations of algebras. Using HOL's type-definition construct with this predicate, we define a new type *alg* for algebras:

```
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)
```

Finally, we define accessor functions $sigAlg, carrierAlg$, and $funsAlg$ that 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))
```

4.2 Σ-Algebras as a Category

A Σ-homomorphism between two Σ-algebras $(\mathcal{A}, I)$ and $(\mathcal{A}', I')$ is a collection of functions $h_s : A_s \rightarrow A'_s$ such that, for a Σ operator ρ with type $s_1 \times \cdots \times s_n \rightarrow s$,

$$h_s(I_\rho(v_1, v_2, ..., v_n)) = I'_\rho(h_{s_1}\ v_1, h_{s_2}\ v_2, \ldots, h_{s_n}\ v_n).$$

Intuitively, this equation says that Σ-homomorphisms preserve the algebraic structure: applying the I-interpretation of the operator ρ to appropriate values $v_1, \ldots, v_n$ and then translating that result to a value in A'_s yields an equivalent result as first translating each of the values v_i to elements of A'_{s_i} and then applying the I'-interpretation of ρ to those values.

For any signature Σ, $\mathbf{Alg}_\Sigma$ is the category whose objects are Σ-algebras and whose arrows are Σ-homomorphisms. The identity arrows are simply those homomorphisms for which each h_s is the identity function, and composition is standard function composition (it is easy to verify that the composition of two homomorphisms is indeed a homomorphism).

In HOL, we define the category $\mathbf{Alg}_\Sigma$ in such way that the signature Σ is taken as a parameter. The 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 the category $\mathbf{Alg}_\Sigma$ is represented as a triple

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

where m and n are the domain and codomain Σ-algebras of the arrow and $h : sort \rightarrow value \rightarrow value$ is a function that maps the elements of m's carrier sets to elements of the corresponding carrier sets of n. The predicate $algHom$ defines a homomorphism between two algebras that have the same signature. The predicate $sigmaAlgA$ picks out $\mathbf{Alg}_\Sigma$ arrows 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 the category $\mathbf{Alg}_\Sigma$ is the function $(\lambda s.I : value \rightarrow value)$. The composition of arrows (a, f, b) and (b, g, c) is defined to be $(a, \lambda s.((g\ s) \circ (f\ s)), c)$.

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)
```

Proving that *algCat* represents a category amounts to proving the following goal:

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

Due to time limitations, we have not yet performed this proof within HOL. However, based on our prior experience with HOL and our knowledge that the underlying category theory is correct, we are confident that this goal could be proven with HOL without siginficant difficulties.

4.3 Algebraic Terms

Algebraic specifications describe abstract data types (ADTs) by augmenting signatures with descriptions of the characteristic properties of the ADTs. These properties of ADTs can be expressed as formulas (such as equations) on the terms of Σ-algebras.

To define the algebraic terms associated with a given signature $\Sigma = (S, \Omega)$, we begin with an infinite set V of symbols called *variables* assumed to be distinct from all the sorts and operator symbols in Σ. A *sort assignment* Γ is a finite set of pairs (x, s), where $x \in V$ is a variable and $s \in S$ is a sort; Γ must be consistent, in that it may associate at most one sort with any particular variable. We then can define by mutual induction a family of sets $\mathit{Terms}(\Sigma, \Gamma) = \{\mathit{Terms}^s(\Sigma, \Gamma) \mid s \in S\}$ as follows, where each set $\mathit{Terms}^s(\Sigma, \Gamma)$ is the collection of *Σ-terms of sort s under the sort assignment* Γ:

- If (x, s) is in Γ, then x is in $\mathit{Terms}^s(\Sigma, \Gamma)$.
- If, for each $i \in \{1, 2, \cdots, n\}$, t_i is a term in $\mathit{Terms}^{s_i}(\Sigma, \Gamma)$, and if ρ is a function symbol of type $(s_1 \times s_2 \times \cdots \times s_n \to s)$, then $\rho(t_1, t_2, \ldots, t_n)$ is a term in the set $\mathit{Terms}^s(\Sigma, \Gamma)$.

A Σ-equation with respect to the sort assignment Γ is a pair of terms (t_1, t_2) such that t_1 and t_2 are both elements of $\mathit{Terms}^s(\Sigma, \Gamma)$, for some sort s. Such an equation is conventionally written as: $t_1 =_s t_2[\Gamma]$.

To define the algebraic terms of a signature Σ in HOL, we first introduce a new base type *variable* to represent our variables. We then represent a sort assignment as a set of pairs $(x, s) : \mathit{variable} \times \mathit{sort}$, where x is a variable and $s \in S$ is its corresponding sort. The HOL predicate *gammaRes* identifies those sets that are valid sort assignments under the signature $\Sigma = (S, \Omega)$.

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

Note that this implementation does not need to verify that each sort assignment Γ is consistent, because each variable v will always appear along with its sort in a pair (v, s). Intuitively, each pair (v, s) can be viewed as representing a variable v_s of sort s, and hence (for example) the pairs (x, int) and (x, bool) represent distinct variables x_{int} and x_{bool}.

We first define a recursive HOL-type *ppreT*, which provides an abstract-syntax representation for Σ-terms and their associated types. Strictly speaking, this new type is slightly more general than our desired Σ-terms, as it will also contain items that technically are only portions of valid Σ-terms. For example, if an operator ρ has type $(s_1 \times s_2 \times \cdots \times s_n) \rightarrow s$ and t_1 is a term of type s_1, then $\rho\ t_1$ is not itself a Σ term. However, it is convenient to allow such partial terms in our abstract syntax, and we shall be able to easily pick out the valid terms from the set of *ppreT* values.

The recursive type *ppreT* has the following three constructors:

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

A value of form $\mathit{Leafv}(x, s)$ represents a variable x of sort s. A value of form $\mathit{Leafo}(op, [s_1, s_2, \cdots, s_n], s)$ represents an operator op of type $(s_1 \times s_2 \times \cdots \times s_n) \rightarrow s$. Finally, a value of $\mathit{Comb}\ t_1\ t_2$ corresponds to a partially instantiated term.

We define a HOL predicate *isPreSigmaTerm* that determines the appropriate type for each element of *ppreT* as follows (*HD* and *TL* return the head and tail of a list):

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

$$\frac{}{\mathit{isPreSigmaTerm\ sigma\ gamma}\ (sl, s)(\mathit{Leafo}\ (op, sl, s))}\ \begin{array}{l} \mathit{inOmega\ sigma}\ f \\ \mathit{gammaRes\ sigma\ gamma} \end{array}$$

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

A Σ-term of type s under Γ is simply an element of *ppreT* whose associated type is $([\,], s)$. In HOL, these terms can be represented as four-tuples $(\Sigma, \Gamma, t, s) : \mathit{sig} \times (\mathit{variable} \times \mathit{sort})\mathit{set} \times \mathit{ppreT} \times \mathit{sort}$. The predicate *isSigmaTerm* identifies those four-tuples that are valid representations of Σ-terms. Using HOL's type-definition construct, we can also introduce a new HOL type *sigmaTm*.

```
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)
```

We define accessor functions *sigSigmaTm*, *gammaSigmaTm*, *tmSigmaTm*, and *tySigmaTm* that 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 Σ terms $(t_1, t_2) : sigmaTm \times sigmaTm$ where t_1 and t_2 are of the same type. The predicate *isSigmaEq* identifies those pairs of Σ-terms that satisfy this constraint, and we use this predicate to define a 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)
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)
```

We introduce accessor functions *leftTermEq* and *rightTermEq* that extract the left and right terms from a Σ-equation. Likewise, we introduce functions *sigSigmaEq*, *gammaSigmaEq*, *ltmSigmaEq*, *rtmSigmaEq*, and *tySigmaEq* that obtain the signature, the variable assignment, the left term, the right term, and the terms' type from an arbitrary Σ-equation.

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

5 Algebraic Specifications

As we have discussed in previous sections, every signature has a collection of models, not all of which necessarily capture our intentions. To reduce this collection to those models that *do* capture the intended meaning, it is necessary to add constraints to the signatures that limit the potential models. These constraints can be represented as equations that are added to a given signature; the result is an *algebraic specification.*

An algebraic specification is a pair (Σ, E), where Σ is a signature and E is a set of Σ-formulas that serves as axioms of the specification. Algebraic specifications can be used to specify computer systems, where Σ describes the interface of the system and E is the desired system properties.

For example, we can define *TL-spec* $= (\Sigma_{tl}, E_{tl})$ as a specification for the traffic light and *BP-spec* $= (\Sigma_{bp}, E_{bp})$ as a specification for the boolean pair, where Σ_{tl} and Σ_{bp} are the signatures described on page 6. With Σ_{tl}, the specification *TL-spec* states that a traffic light has exactly one of the three distinct colors. With Σ_{bp}, the specification *BP-spec* states that a boolean pair has exactly one of the four distinct values.

E_{tl} :

$\underline{\text{color-distinct}}$: $(green \neq yellow) \wedge (yellow \neq red) \wedge (red \neq green)$

$\underline{\text{color-cases}}$: $for\ each\ color\ x, (x = green) \vee (x = yellow) \vee (x = red)$

E_{bp} :

$\underline{\text{boolPair-distinct}}$:

$$(TT \neq TF) \wedge (TT \neq FT) \wedge (TT \neq FF) \wedge (TF \neq FT) \wedge (TF \neq FF) \wedge (FT \neq FF)$$

$\underline{\text{boolPair-cases}}$: $for\ each\ boolPair\ y, (y = TT) \vee (y = TF) \vee (y = FT) \vee (y = FF)$

A Σ-algebra (A, I) is a model of a specification (Σ, E) provided that (A, I) satisfies all the formulas in E. That is, for every possible variable assignment, all the Σ-formulas in E hold with respect to the carrier-set assignment A and the function-symbol assignments I.

5.1 Specification as a HOL Type

In HOL, we represent specifications by triples (Σ, Γ, E) of type $sig \times (variable \times sort)\ set \times sigmaEq\ set$, so that Σ is a signature, Γ is a variable assignment, and E is a set of Σ-equations.

The predicate *isSpec* identifites the subset of these triples that correspond to valid algebraic specifications.

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

Using this predicate along with HOL's type-definition construct, we can then define algebraic specifications as a new HOL type *spec*.

```
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)
```

As before, we also define accessor functions *sigSpec, gammaSpec,* and *ESpec* that 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))
```

5.2 Algebraic Specifications as a Category

The category **Spec** is the category of algebraic specifications: its objects are specifications, and its arrows are specification morphisms. A specification morphism f between two specifications (Σ, E) and (Σ', E') is a signature morphism between Σ and Σ' that preserves theorems: f takes any axiom (i.e., Σ-formula) in E either to an axiom in E' or to a theorem deducible from the axioms in E'.

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

An arrow in the category **Spec** is therefore represented as a triple $(d, (fs, fo, fv), c)$ with the following type:

$$
\begin{array}{l}
spec \times \\
((sort \rightarrow sort) \times \\
\quad ((operator \times sort\ list \times sort) \rightarrow (operator \times sort\ list \times sort)) \\
\quad ((variable \times sort) \rightarrow (variable \times sort))) \times \\
spec
\end{array}
$$

As before, we define accessor functions that retrieve the various components of 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 the variable-assignment mapping $f_v : \Gamma \rightarrow \Gamma'$ determine the transformation from the Σ-equations of one specification to the Σ-equations of another specification. The terms of Σ (under Γ) can be transformed to terms of Σ' (under Γ') in the obvious inductive fashion:

- Each (sub)term of form $(Leafv\ (x, s))$ is transformed to the term $Leafv\ (fv\ (x, s))$.
- Each (sub)term of form $(Leafo\ (op, sl, s))$ is transformed to the term $Leafo\ (fo\ (op, sl, s))$.
- Each (sub)term of form $(Comb\ t1\ t2)$ is transformed by transforming its components $t1$ and $t2$.

The recursive function *transPPT* is formalized in HOL as follows:

```
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))
```

The Σ'-types of the newly constructed terms are obtained by applying the mapping f_s to the original Σ-type of a term, so that each Σ-term t of type s is transformed to term t' with 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))
```

Finally, these transformations can be applied componentwise to transform a Σ-equation to a corresponding Σ'-equation, as follows:

```
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)))
```

Because specification morphisms must preserve theorems, we need a way to verify that each Σ-equation is translated to an equation derivable from the Σ'-equations. Ideally, we would define a function *thmsDerivable* : *sigmaEq set* $\rightarrow$ *sigmaEq set* that constructs the set of all theorems derivable from a given set of axioms. We could then define a specification arrow in HOL as follows, where the predicate *inGammaSpec* defines the elements that are in the variable-assignment set Γ of a specification and the 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)
```

This approach is clearly not feasible, as it is impossible to construct the set that contains all the theorems derivable from a given set of axioms. In fact, it is generally undecidable whether a given formula is a consequence of a collection of axioms.

Based on our limited experience, however, we believe that in practice a user is capable of ensuring that the mapping preserves theorems. The specification morphisms used to refine specifications in practice tend to introduce new constraints without significant renaming or significant omissions (i.e., Σ-axioms tend to be translated to Σ'-axioms). Furthermore, for more complicated translations, a user could prove the necessary preservations separately using the HOL theorem prover. Having verified the necessary conditions for the given specifications, the user could then introduce a HOL definition that provides a sufficient approximation to the set `thmsDerivable(ESpec c)`, namely a set that contains precisely the (finite number of) verified axiom translations. Although such a process is not completely automated, it does allow a user to verify the validity of the translation and to generate assured specifications and refinements.

Once specification arrows have been defined, we introduced a 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 the object (Σ, E) in the category **Spec** is the identity arrow of the object Σ in the category **Sig**, and composition in **Spec** is defined in the same way as in the 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)
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)
```

Finally, the tuple *specCat* could be proved to represent a category by proving the following goal.

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

6 Summary and Future Work

Throughout this report, we have identified many of the categories and constructs underlying algebraic specifications and their interpretations. Furthermore, we have formulated them in higher-order logic and (in most cases) verified the correctness of our formulation.

The purpose of computer-assisted reasoning is to provide to help nonexperts in a given domain to nonetheless have confidence in their analysis. In this work, we have not uncovered new uses of category theory or proved new theorems about category theory. Instead, we have embedded category theory in a form that nonexperts can use in the future to construct assured specifications and ultimately assured code. The objective of our formulation of category theory in HOL is to fully explicate the underlying principles of construction that algebraic specifications provide.

At present, this work remains incomplete. As discussed in Section 5, verifying that a specification morphism is valid introduces additional proof obligations for the user. We have not yet investigated the various mechanisms for integrating these obligations into the system. We would like to better understand the trade-offs involved and how well these mechanisms work in practice.

In addition, we have not yet implemented the categorical mechanisms that underlie the composition of specifications or refinements of specifications. This type of composition and the notions of refinement rely on categorical *pushouts* (or, more generally, *colimits*), which provide a canonical way to compose specifications. Intuiviely, a specification morphism f from A to B indicates how B can be viewed as adding additional constraints to A. The existence of pushouts in the category **Spec** assures that whenever a specification A can be further constrained by two different specifications B and C (via morphisms f and g), that there is a canonical specification D that captures precisely the additional constraints imposed by both B and C. Given such specification morphisms $f : A \rightarrow B$ and $g : A \rightarrow C$, the pushout can be constructed algorithmically, and hence we do not anticipate any significant difficulties formulating pushouts in HOL.

References

[1] J. A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R.T. Yeh, editor, *Current Trends in Programming Methodology, Volume IV*, pages 80–149. Prentice Hall, 1978.

[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] Kestrel Institute, 3260 Hillview Ave, Palo Alto, CA. *Specware Language Manual*, 1.02 edition, June 1995.

[4] J. Linn. Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures. RFC 1421, DEC, February 1993. ftp: ds.internic.net.

[5] Lockwood Morris. Interim Partial Description of a Representation for Categories in HOL. Communicated through private channel, 1998.

[6] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists.* The MIT Press, Cambridge, Massachusetts, 1991.

[7] Sten Agerholm. Experiments in formalizing basic category theory in higher order logic and set theory. *http://www.cs.chalmers.se/ ilya/FMC/*, 1995.

[8] Dan Zhou. *High-Confidence Development of Secure E-mail Systems.* PhD thesis, Syracuse University, 1999.

[9] Dan Zhou and Shiu-Kai Chin. Verifying Privacy Enhanced Mail Functions with Higher Order Logic. *Network Threats, DIMACS Series in Discrete Mathematics*, 38:11–20, 1998.

[10] Dan Zhou, Joncheng C. Kuo, Susan Older, and Shiu-Kai Chin. Formal Development of Secure Email. In *Proceedings of the 32nd Hawaii International Conference on System Sciences*, January 1999.

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.