Abstract Theories in HOL

Phillip J. Windley

University of Idaho Laboratory for Applied Logic, Moscow ID 83843

Abstract
Abstract theories are widely used in mathematics because the provide a convenient way to reason about classes of structures. Abstract theories provide proofs about abstract structures which can then be used to reason about specific instances of those structures. Abstract theories are useful for specifying and verifying computer systems because they provide structure and guidance in the specification and verification processes and provide a convenient method of theorem reuse. This report describes and documents a package for using abstract theories in the HOL theorem proving system.

Keyword Codes: F.4.1; F.4.3
Keywords: Mathematical Logic; Formal Languages

1 Introduction

A theory is a set of types, definitions, constants, axioms and parent theories. Logics are extended by defining new theories. An abstract theory is parameterized so that some of the types and constants in the theory are undefined inside the theory except for their syntax and an algebraic specification of their semantics. Group theory provides an example of an abstract theory: the multiplication operator is undefined except for its syntax (a binary operator on type `:group`) and a semantics given by the axioms of group theory.

Abstract theories are useful because they provide proofs about abstract structures which can then be used to reason about specific instances of those structures. In groups, for example, after showing that addition over the integers satisfies the axioms of group theory, we can use the theorems from group theory to reason about addition on the integers.

This report describes the use of abstract theories in the HOL theorem prover [Gor88]. The current version of the abstract theory package has evolved greatly since the original report was published [Win90].

2 Abstract Theories.

There are two key components of an abstract theory: (1) the *abstract representation* and (2) the *theory obligations*. The *abstract representation* is a set of abstract objects and a set of abstract operations. The operations are unspecified; that is, we don't know (inside the theory) what the objects and operations mean. Their meaning is specified through the theory obligations: a set of predicates that define relationships among members of the abstract representation. The abstract theory describes a model. Any structure with objects and operations that satisfy the predicates is a homomorphism of that model.

The theory obligations axiomatize the theory. Using the obligations as axioms, we prove theorems of interest about the abstract objects and operations. The goal is to use the abstract theory to reason about specific objects by instantiating the abstract theory with a concrete representation which has been shown to meet the obligations. The instantiation specializes the abstract theorems, producing a set of theorems about the concrete representation. The concrete representation is an instance of the abstract theory and represents a *member* of the class of abstract objects that it describes.

HOL, the verification environment used in the research reported here, does not explicitly support abstract theories; however, HOL's metalanguage, ML, combined with higher–order logic, provides a framework sufficient for implementing abstract theories. Several specification and verification systems, such as OBJ and EHDM, offer explicit support for abstract theories. We briefly describe them for comparison.

OBJ. OBJ is a specification and programming language developed by Joseph Goguen *et al.* that has most recently been described in [GW88]. OBJ is widely known and the semantics of its theories and views match our use of abstract theories. OBJ is based on a many–sorted (or typed) algebraic semantics and supports parameterized specification and programming [Gog84]. OBJ has three kinds of entities:

1. **Objects**, which are concrete modules that encapsulate executable code,

2. **Theories**, which are parameterized modules that correspond to abstract theories as used in this report, and

3. **Views**, which bind objects and theories to parameters in another theory.

Objects are said to contain executable code because the expressions in an object module give the initial algebraic semantics of the sorts and operations being defined. The fact that their semantics is initial implies that they describe just one model (up to isomorphism). Theories, on the other hand, are said to have a "loose" semantics since they define a variety of models. A loose semantics describes a class of objects; any member of that class will satisfy the theory.

A view is *not* an instantiation. Instantiation is done using a special command, **make**, after the view has been established. A view can be seen as a mapping of the operators and objects from one module onto a theory, as well as a declaration of intent that the module meets the obligations set forth in the equations of the theory module. OBJ does not require that the user prove that the obligations are met—a simple declaration is sufficient. Of course, if the view is not proper, then the OBJ program will not operate as intended.

EHDM. EHDM is a specification and verification system developed by SRI International [EHD88]. The language of EHDM is based on first–order predicate logic, but includes some elements of higher–order logic as well. For example, variables can range over functions, functions can return other functions, and functions can appear in quantifications. Parameterized modules are an important part of the EHDM language where they are used to organize specifications. Modules can be parameterized with types, constants, and functions. The module parameters can have constraints placed on them that must be met before the module can be instantiated.

In EHDM, a parameterized module is called a *generic module* and an instantiation is called a *module instance*. EHDM module declarations give the uninterpreted types, constants, and functions over which the module is parameterized. This declaration is analogous to our abstract representation.

The module body contains (among other things) an **ASSUMING** clause that gives the properties of the module parameters. The formulae in the **ASSUMING** clause are analogous to our theory obligations.

The module can also contain declarations of concrete types, constants, and functions that define the theory associated with the module and proofs of theorems about the abstract operations in the theory. These proofs may rely on the formulae in the **ASSUMING** clause.

3 Using the Abstract Theory Package.

This section briefly describes the major functions in the abstract theory package. The following section provides an example of its use. Complete descriptions of all the commands are given in the user's manual that accompanies the package. For an example of how abstract theories can be used in computer system specification and verification see [Win91, Win92].

Before beginning an abstract theory, the ML file `abs_theory` must be loaded. This defines the commands in the abstract package and modifies some of the standard HOL commands to support its operation.

One declares a new abstract theory in the same way that one declares a standard theory, using `new_theory`. One is free to use any of the standard HOL commands for manipulating a draft theory in their usual manner. For example, definitions are made in the usual way using `new_definition`.

3.1 Abstract Representations.

The abstract representation describes the abstract objects and operators in the abstract theory. The abstract theory package defines `new_abstract_representation` for declaring the abstract representation. The function is applied to a string representing the name of the abstract object and a list of string–type pairs. The first member of the pair gives the name of the abstract operator and the second member of the pair is the its type. Any number of abstract operators can be defined in an abstract representation. One can use `new_abstract_representation` more than once in a single theory defining more than one abstract object.

The system does not require that abstract objects be specifically declared. We represent abstract objects as type variables in HOL (denoted by a prepended asterisk). Since HOL does not require that type variables be declared, we are free to use them wherever we wish. The declaration of abstract objects is implicit, being the set of type variables occurring in the abstract representation.

The result of declaring a new abstract representation is a theorem that can be saved to use with `abs_type_info` to retrieve the type of the abstract object when this is difficult to otherwise discern.

When one defines a constant in the abstract theory, by convention, the first argument to the constant will be a variable with the same type as the abstract representation. This variable must, in turn be the first argument to any of the abstract constants from the abstract representation used in the definition. Later, during instantiation, the definition will be applied to a concrete representation and the instantiation functions will replace the abstract constants with the appropriate concrete constants in the instantiation.

3.2 Theory Obligations.

The theory obligations are declared using the ML function `new_theory_obligations`. The function takes a string–term pair as its sole argument. The pair represents a theory obligation, giving the name and the predicate defining the theory obligation. The pair can be thought of as an axiom defining the semantics of the abstract objects.

The predicate is usually a conjunction of obligations. These obligations will be available for use in the draft theory. The system will automatically add them to the assumption list when the HOL commands for declaring abstract goals and proving abstract theorems, such as `set_abs_goal` and `prove_abs_thm`, are used. The HOL command `close_theory` closes the current draft and flushes the theory obligations.

3.3 Instantiating Theories.

One makes use of an abstract theory by instantiating it. The first step is to make the abstract theory a parent of the draft theory using the ML function `new_abstract_parent`.

HOL theories differentiate between definitions and theorems and so there is an ML function for instantiating each. Instantiating any abstract object requires that we create a concrete representation. Concrete representations are created by applying the name of the abstract object to the concrete objects that are to be used for the instantiation.

Abstract Definitions. Abstract definitions are definitions which use the abstract objects from the abstract representation. By convention, the first variable in an abstract definition is the representation variable and has the same type as the abstract representation. Creating a concrete definition from an abstract one requires two steps:

1. Make an auxiliary definition that uses the abstract definition and applies it to a concrete representation.

2. Use the ML function `instantiate_abstract_definition` to produce a concrete instance of the abstract definition from the auxiliary definition.

The result of this instantiation is a theorem that defines a concrete instance of the abstract definition and makes no reference to the abstract definition.

Abstract Theorems. In drafting an abstract theory, one normally proves theorems about the abstract representation using the theory obligations as axioms. When the abstract theory is used, we instantiate the theorems in it so that the theory obligations are discharged and the new concrete theorems stand on their own.

The ML function `instantiate_abstract_theorem` instantiates one abstract theorem. The function takes four arguments:

1. The name of the abstract theory where theorem reside.
2. The name of the abstract theorem to instantiate.
3. A list of term pairs that instantiate variables with concrete representations. The first term in the pair is the variable to instantiate and the second is the concrete representation.
4. A list of theorems that satisfy the theory obligations in the subject abstract theorem. These theorems discharge the antecedents of the abstract theorems.

The new theorem resulting from the instantiation is *not* automatically saved in the current theory, but must be explicitly saved using `save_thm`.

4 Example: Group Theory

This section demonstrates the major features of the abstract theory package. We begin by defining an abstract theory for *monoids*. The following section uses the theory of monoids to create a theory of *groups*. The last section instantiates the theory of groups using exclusive–or as the group operator.

4.1 Defining Monoids

A monoid is an algebra with a binary operator and an identity element. The operator is associative and operating on the identity element with any object in the algebra, x yields that same object, x.

We begin the the session by loading the abstract theory package and entering draft mode:

```
loadf `abs_theory`;;

new_theory `monoid_def`;;
```

The abstract representation is declared using `new_abstract_representation`. We declare the binary operator, `op`, and an identity element, `e`:

```
let MONOID = new_abstract_representation `monoid`
   [
    (`op`,":* -> * -> *")
   ;
    (`e`,":*")
   ];;
```

The abstract representation gives only the name and type of the operators.

A declaration of an abstract representation will almost always be followed immediately by a declaration of the theory obligation since the abstract representation introduces the structure of the abstract entity, but not the semantics. The theory obligations declare the semantics (or axioms) for the abstract entity. The theory obligations are contained in a predicate that states the required properties of `op` and `e`:

```
new_theory_obligations
   (`IS_MONOID`,
    "∀m:(*)monoid . IS_MONOID m =
        (∀x:* .op m x (e m) = x) ∧
        (∀x:* .op m (e m) x = x) ∧
        (∀ x y z:*. (op m x (op m y z)) = (op m (op m x y) z))");;
```

Note that when `op` and `e` are used in a term, they always take as their first argument a variable with the type of the abstract object. While this is an implementational requirement, one can think of it as identifying `op` and `e` as abstract operators and giving the specific abstract object of which they are part. This last point is important since some theorems may use more than one abstract object and the abstract operators are differentiated by their argument (as shown in the example below).

We will begin with an interactive proof that the identity element is unique. We place the goal on the goal stack in the usual manner:

```
g
"∀ (m:(*)monoid) (f:*) .
       (∀(a:*) .(op m a f = a) ∧ (op m f a = a)) ⇒  (f = (e m))";;
```

HOL responds with the goal. Note that the theory obligations are automatically placed on the assumption list.

```
"∀f. (∀a. (op m a f = a) ∧ (op m f a = a)) ⇒  (f = e m)"
    [ "∀x y z. op m x(op m y z) = op m(op m x y)z" ]
    [ "∀x. op m(e m)x = x" ]
    [ "∀x. op m x(e m) = x" ]
```

We begin by stripping the universally quantified variables and the antecedent of the implication:

```
e(
REPEAT STRIP_TAC
);;

OK..
"f = e m"
    [ "∀x y z. op m x(op m y z) = op m(op m x y)z" ]
    [ "∀x. op m(e m)x = x" ]
    [ "∀x. op m x(e m) = x" ]
    [ "∀a. (op m a f = a) ∧ (op m f a = a)" ]
```

We can specialize the antecedent that we just placed on the assumption list with the identity element. Either conjunct can be used to solve the goal, we'll use the first.

```
e(
ASSUME_TAC (
  (CONV_RULE SYM_CONV) (
  CONJUNCT1 (
  SPEC "e (m:(*)monoid)" (
  ASSUME "∀a. (op m a f = a) ∧ (op m f a = a)"))))
);;

OK..
"f = e m"
    [ "∀x y z. op m x(op m y z) = op m(op m x y)z" ]
    [ "∀x. op m(e m)x = x" ]
    [ "∀x. op m x(e m) = x" ]
    [ "∀a. (op m a f = a) ∧ (op m f a = a)" ]
    [ "e m = op m(e m)f" ]
```

The result of the last step can be substituted into the goal:

```
e(
SUBST1_TAC ASSUME "e m = op m(e m)f"
);;

OK..
"f = op m(e m)f"
    [ "∀x y z. op m x(op m y z) = op m(op m x y)z" ]
    [ "∀x. op m(e m)x = x" ]
    [ "∀x. op m x(e m) = x" ]
    [ "∀a. (op m a f = a) ∧ (op m f a = a)" ]
    [ "e m = op m(e m)f" ]
```

The result can be rewritten with the assumptions to solve the goal:

```
e(
ASM_REWRITE_TAC []
);;

OK..
goal proved
. ⊢ f = op m(e m)f
.. ⊢ f = e m
.. ⊢ f = e m
. ⊢ ∀f. (∀a. (op m a f = a) ∧ (op m f a = a)) ⇒  (f = e m)

Previous subproof:
goal proved
```

The interactive proof can be packaged up as a proof script:

```
let IDENTITY_UNIQUE = ABS_TAC_PROOF
   (([],"∀ (m:(*)monoid) (f:*) .
       (∀(a:*) .(op m a f = a) ∧ (op m f a = a)) ⇒  (f = (e m))"),
    REPEAT GEN_TAC
    THEN STRIP_GOAL_THEN (⊢ .
        SUBST1_TAC (
        SYM_RULE (
        CONJUNCT1 (
        SPEC "e (m:(*)monoid)" thm))))
    THEN ASM_REWRITE_TAC []
   );;


IDENTITY_UNIQUE =
. ⊢ ∀f. (∀a. (op m a f = a) ∧ (op m f a = a)) ⇒  (f = e m)
```

The function ABS_TAC_PROOF is like TAC_PROOF except that when used on a goal with an abstract variable, the theory obligations are added to the assumption list before the proof is started. If there is more than one abstract variable the theory obligations are added to the assumption list for each. The fact that the theory obligations are needed to establish the theorem and are on the assumption list obviously influences the style of proof. An example in the next section demonstrates how to explicitly declare the obligations.

The next theorem gives an example that uses two abstract objects. The theorem proves that if the operations for two monoids are the same then their identity elements must be the same as well.

```
let OP_DETERMINES_IDENTITY = ABS_TAC_PROOF
   (([],
     "∀ m1 (m2:(*)monoid) . (op m1 = (op m2)) ⇒  (e m1 = (e m2))"),
    REPEAT STRIP_TAC
    THEN let t1 = ASSUME "∀x:*. op m1 (e m1) x = x" in
         SUBST_TAC (map SYM_RULE [SPEC "e m2:*" t1])
    THEN let t2 = ASSUME "∀x:*. op m2 x (e m2) = x" in
         SUBST_TAC (map SYM_RULE [SPEC "e m1:*" t2])
    THEN ASM_REWRITE_TAC []
  );;


OP_DETERMINES_IDENTITY = .. ⊢ (op m1 = op m2) ⇒  (e m1 = e m2)
```

In this example `m1` and `m2` are two distinct abstract objects. The term `(op m1)` represents the binary operation of monoid `m1` and the term `(op m2)` represents the binary operation of monoid `m2`. When there is more than one abstract object in a goal, the abstract theory package places theory obligations on the assumption list for each. Note that the style of proof in this example is more goal oriented than the last example even though, as in the last example, the goal is proven using the assumptions.

4.2 Defining Groups

A group is a monoid extended with an inverse operator defined such that when the binary operator is applied to any element and its inverse, the identity element is the result.

To define a theory of groups, we take care of the usual front matter and also declare `monoids_def` to be an abstract parent since we are going to establish some properties of groups from our theory of monoids.

```
loadf `abs_theory`;;

new_theory `group_def`;;

new_abstract_parent `monoid_def`;;
```

The call to `new_theory` clears the global theory obligation list. `new_abstract_parent` puts the theory obligations from `monoid_def` on the theory obligation list.

The next step is to define the abstract representation:

```
let GROUP = new_abstract_representation `group`
   [
    (`fn`,":* -> * -> *")
   ;
    (`id`,":*")
   ;
    (`inv`,":* -> *")
   ];;
```

Note that we do not simply define the abstract representation for groups as an extension to the representation for monoids. Without subtypes, the implementation and use of extensions becomes too unwieldy. Also, since HOL does not allow operator overloading, we had to give the operator and identity element for groups different names from the operator and identity element for monoids.

The theory obligations are those for monoids extended with two additional facts for the inverse function:

```
new_theory_obligations
    (‘IS_GROUP‘,
     "∀g:(*)group . IS_GROUP g =
        (∀x:* .fn g x (id g) = x) ∧
        (∀x:* .fn g (id g) x = x) ∧
        (∀x:* .fn g x (inv g x) = (id g)) ∧
        (∀x:* .fn g (inv g x) x = (id g)) ∧
        (∀ x y z:*. (fn g x (fn g y z)) = (fn g (fn g x y) z))");;
```

Again, because HOL does not have a notion of subtypes, we do not define the theory obligations for groups in terms of the theory obligations for monoids.

The connection between groups and monoids is established by the following trivial theorem:

```
let GROUP_EXTENDS_MONOID = ABS_TAC_PROOF
   (([],
     "∀ g:(*)group . IS_MONOID(monoid (fn g) (id g))"),
    EXPAND_THOBS_TAC ‘monoid_def‘
    THEN ASM_REWRITE_TAC []
   );;

GROUP_EXTENDS_MONOID = ... ⊢ IS_MONOID(monoid(fn g)(id g))
```

The theorem states that the operator and identity element for groups can be used in place of the operator and identity element for monoids. The tactic EXPAND_THOBS_TAC is used to expand the theory obligations for monoids as the first step of the proof. The result is an abstract theorem since it relies on the theory obligations of group_def.

We can make use of the fact that a group is a monoid by instantiating the theorem that the identity is unique from monoid_def:

```
let IDENTITY_UNIQUE =
    instantiate_abstract_theorem ‘monoid_def‘ ‘IDENTITY_UNIQUE‘
        ["m","monoid (fn (g:(*)group)) (id (g:(*)group))"]
        [GROUP_EXTENDS_MONOID];;

IDENTITY_UNIQUE =
... ⊢ ∀f. (∀a. (fn g a f = a) ∧ (fn g f a = a)) ⇒  (f = id g)
```

We instantiate the abstract representation variable "m" with a concrete (from the standpoint of monoid_def) representation made from the operator and identity element of group_def. The theorem that a group extends a monoid is used to discharge the theory obligations for monoids. Note that the instantiated theorem must be explicitly saved on the current theory.

The resulting theorem is really a theory about groups, not monoids, as can be seen from the following expression which checks the type of the abstract representation variable "g" in IDENTITY_UNIQUE:

```
(type_of o hd o frees o concl o DISCH_ALL) IDENTITY_UNIQUE;;

":(*)group" : type
```

In addition to instantiating theorems from monoid_def, we can prove theorems about groups directly. For example, we can prove that left cancellation holds for groups:

```
let LEFT_CANCELLATION = prove_abs_thm
   (`LEFT_CANCELLATION`,
     "∀ (g:(*)group) (x y a:*) .
      ((fn g) a x = ((fn g) a y)) ⇒  (x = y)",
    REPEAT STRIP_TAC
    THEN ACCEPT_TAC (
      let t1 = (ASSUME "∀x y z. fn g x(fn g y z) = fn g(fn g x y)z") and
          t2 = (ASSUME "∀x. fn g(inv g x)x = id g") and
          t3 = (ASSUME "∀x. fn g(id g) x  = x") and
          t4 = (ASSUME "fn g a x = fn g a y") in
      SYM_RULE (
      REWRITE_RULE [t1;t2;t3] (
      REWRITE_RULE [t2;t3;t4] (
      ISPECL ["(inv g a)";"a";"x"] t1))))
   );;

LEFT_CANCELLATION = ... ⊢ ∀x y a. (fn g a x = fn g a y) ⇒  (x = y)
```

In this case, we used prove_abs_thm, so the the resulting theorem is explicitly saved in the current theory.

Because some people may be uncomfortable with implicit theory obligations and their automatic inclusion in the assumptions, the abstract theory package supports explicit theory obligations as well. For example, consider the following proof that the inverse function is reversible (that is that the inverse operator applied twice is the identity function):

```
let INVERSE_INVERSE_LEMMA = prove_thm
   (`INVERSE_INVERSE_LEMMA`,
    "∀(g:(*)group) . IS_GROUP(g) ⇒  ∀a .(((inv g) ((inv g) a)) = a)",
    STRIP_THOBS_TAC
    THEN GEN_TAC
    THEN ACCEPT_TAC (
        let t1 = ASSUME "∀x. fn g x(inv g x) = id g" and
            t2 = ASSUME "∀x. fn g (inv g x)x = id g" in
        let LC_LEMMA =
              ISPECL ["inv g (inv g a)";"a";"inv g a"] LEFT_CANCELLATION in
        MATCH_MP LC_LEMMA
                 (TRANS (ISPEC "(inv g) a" t1)
                        (SYM_RULE (ISPEC "a" t2))))
   );;

INVERSE_INVERSE_LEMMA = ⊢ ∀g. IS_GROUP g ⇒  (∀a. inv g(inv g a) = a)
```

In this case, we use the standard prove_thm function and so the theory obligations were not automatically added to the assumption list. The theory obligation are given explicitly as the antecedent to an implication. The tactic STRIP_THOBS_TAC is a convenient way to strip explicit theory obligations from the goal and expand them on the assumption list.

We can use a different proof style using the theorem continuation STRIP_THOBS_THEN for the same theorem:

```
let ALTERNATE_INVERSE_INVERSE_LEMMA = TAC_PROOF
   (([],
    "∀(g:(*)group)  . IS_GROUP(g) ⇒  ∀ a .(((inv g) ((inv g) a)) = a)"),
    STRIP_THOBS_THEN (λ thm .
        let thl = CONJUNCTS thm in
        MAP_EVERY ASSUME_TAC thl THEN
        GEN_TAC THEN
        MATCH_MP_TAC
         (ISPECL ["inv g (inv g a)";"a";"inv g a"] LEFT_CANCELLATION) THEN
        REWRITE_TAC thl)
  );;

ALTERNATE_INVERSE_INVERSE_LEMMA =
   ⊢ ∀g. IS_GROUP g ⇒  (∀a. inv g(inv g a) = a)
```

The only caveat to using STRIP_THOBS_THEN is that the accompanying theorem–tactic must put the theory obligations on the assumption list if any other abstract theories are to be used. Since we use LEFT_CANCELLATION, the tactic MAP_EVERY ASSUME_TAC thl is used to add the theory obligations to the assumption list.

4.3 Using Groups

We can instantiate the theory of groups using exclusive–or as the operator, false as the identity element, and the identity function as the inverse operator. As usual, we load the abstract theory package, enter draft mode, and declare groups_def an abstract parent.

```
loadf `abs_theory`;;

new_theory `example`;;

load_library `taut`;;

new_abstract_parent `group_def`;;
```

Before we can instantiate the theorems of group_def we must show that our proposed instantiation discharges the theory obligations:

```
let GROUP_THOBS = TAC_PROOF
   (([],"IS_GROUP(group (λx y.¬(x=y)) F I)"),
    EXPAND_THOBS_TAC `group_def`
    THEN BETA_TAC
    THEN REWRITE_TAC [I_THM]
    THEN TAUT_TAC
   );;
```

The term (group (λ x y. ¬(x=y)) F I) creates the concrete representation.

Using instantiate_abstract_theorem, we can instantiate the theorem of group theory for our concrete representation. The abstract variable "g" is instantiated with our concrete representation.

```
instantiate_abstract_theorem ‘group_def‘ ‘IDENTITY_UNIQUE‘
    ["g","group (λx y.¬(x=y)) F I"]
    [GROUP_THOBS];;


⊢ ∀f. (∀a. (¬(a = f) = a) ∧ (¬(f = a) = a)) ⇒  ¬f


instantiate_abstract_theorem ‘group_def‘ ‘LEFT_CANCELLATION‘
    ["g","group (λx y.¬(x=y)) F I"]
    [GROUP_THOBS];;


⊢ ∀x y a. (¬(a = x) = ¬(a = y)) ⇒  (x = y)


instantiate_abstract_theorem ‘group_def‘ ‘INVERSE_INVERSE_LEMMA‘
    ["g","group (λx y.¬(x=y)) F I"]
    [GROUP_THOBS];;


⊢ ∀a. I(I a) = a
```

The instantiation is done in the same way regardless of whether the theory obligations for the abstract theorems were explicitly stated or not.

5 Conclusions.

Abstract theories are widely used in mathematics because the provide a convenient way of reasoning classes of structures. Abstract theories are useful for specifying and verifying computer systems because they provide structure and guidance in the specification and verification processes and provide a convenient method of theorem reuse. The package we have described shows how abstract theories can be easily used in HOL.

References

[EHD88] SRI International Computer Science Laboratory. *EHDM Specification and Verification System: User's Guide,* Version 4.1, 1988.

[Gog84] J. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, SE-10(5):528–543, September 1984.

[Gor88] Michael J.C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwhistle and P.A Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*. Kluwer Academic Press, 1988.

[GW88] J. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, August 1988.

[Win90] Phillip J. Windley. A poor man's implementation of abstract theories. Technical Report CSE-90-06, University of California, Davis, Division of Computer Science, 1990.

[Win91] Phillip J. Windley. Abstract hardware. In *Proceedings of the ACM/SIGDA International Workshop in Formal Methods in VLSI Design*, January 1991.

[Win92] Phillip J. Windley. A theory of generic interpreters. Technical Report LAL-92-06, University of Idaho, Laboratory for Applied Logic, April 1992.