A Microkernel for a Mechanized Mathematics System

William M. Farmer and Martin v. Mohrenschildt

Department of Computing and Software
McMaster University, 1280 Main Street West
Hamilton, Ontario L8S 4L7, Canada
{wmfarmer,mohrens}@mcmaster.ca

Abstract. A microkernel system is described that provides services for creating and extending formal languages, theories, computations, deductions, and interpretations of one theory in another. In analogy to an operating system microkernel, this microkernel is a platform for implementing multiple logics and interpretations between logics. It is intended to be the bottom layer of a mechanized mathematics system that supports the full mathematics process with the capabilities of both contemporary theorem proving systems and computer algebra systems.

Keywords: Theorem proving, computer algebra, logic implementations.

1 Introduction

Mathematics is a process of creation, exploration, and connection. It consists of three intertwined activities:

1. *Model creation.* Mathematical models representing mathematical aspects of the world are created.
2. *Model exploration.* The models are explored by stating and proving conjectures and by performing calculations.
3. *Model connection.* The models are connected to each other so that results obtained in one model can be used in other models.

With the help of formal logic and the computer, many aspects of the mathematics process can be *mechanized*. Mathematical models and their constituents—entities such as numbers, functions, theorems, proofs, etc. and mathematical operations used in proving, computing, and connecting—can be represented and electronically stored as formal objects. The formal operations can then be applied to the formal entities mechanically. By mechanizing parts of the mathematics process in this way, the process as a whole can become easier to do and the results produced can be more reliable.

A *mechanized mathematics system* (MMS) is a computer environment for doing mathematics in which parts of the mathematics process have been mechanized. An MMS is intended to support, improve, and automate the mathematics process. There are two major types of MMSs today: *theorem proving systems* for proving conjectures and *computer algebra systems* for performing computations. Each type has its own strengths and weaknesses. Theorem proving systems are based on well-defined logical foundations, but are difficult to use and often cannot effectively perform routine computations. Computer algebra systems are relatively fast and easy to use but are not rigorously based and are narrow in scope. Neither type supports the full mathematics process.

Theorem proving systems emphasize the conjecture proving part of the mathematics process. Mathematical models are usually formalized as axiomatic theories expressed in a formal logic. In nearly all theorem proving systems the formal logic is fixed,[1] and in many systems only one axiomatic theory is supported. Very few systems provide support for connecting axiomatic theories in the same logic,[2] and there is currently no system in which axiomatic theories expressed in different logics can be formally connected. Relative to computer algebra systems, theorem proving systems provide very poor support for performing computations.

With the exception of computation, computer algebra systems ignore most parts of the mathematics process. Many systems only support computations in one model, the model of complex arithmetic. There is usually very little support for directing computations by assumptions or for applying in a concrete context results developed in more abstract contexts. Conjecture proving, if it exists at all in a system, is something added on as an afterthought.

An MMS that supports the full mathematics process and that is accessible to a wide range of mathematics practitioners will revolutionize how mathematics is learned and practiced. It will extend the mathematical reach of engineers and scientists, allowing them to use more mathematics in their work and to use it better. It will enable university students to learn mathematics by actively participating in the mathematics process. And it may even change the way mathematicians conduct research.

The development of an MMS that supports the full mathematics process is one the leading problems today of the fields of automated reasoning

[1] The most notable exception is the Isabelle generic theorem proving system [10] in which conjecture proving can be performed in a broad range of logics.

[2] The Ergo [9] and IMPS [4, 5] theorem proving systems allow theories to be connected with each other via interpretations.

and symbolic computation. Such an MMS must include the capabilities of both theorem proving and computer algebra systems.[3] It must also allow multiple logics, multiple theories in each logic, conjecture proving and computation in each theory, and connections between theories so that results developed in one theory can be shared with other theories both in the same logic and in other logics.

This problem is analogous to the problem of how to develop an operating system on which programs written for different traditional operating systems like Unix and Windows can run and interact. One approach is to develop an operating system microkernel that provides only the most essential services of an operating systems such as process management and memory management. Traditional operating systems can then be implemented as applications that run on top of the microkernel.

This paper presents a microkernel for a mechanized mathematics system that is intended to support the full mathematics process. The microkernel provides the functionality needed to implement one or more logics. It consists of a number of different kinds of objects and a number of operations for creating and extending the objects. The microkernel objects and operations are organized into three "services": the Language Service for formal languages (defined in section 2), the Theory Service for theories and interpretations between theories (section 3), and the Derivation Service for computations and deductions (section 4).

A microkernel implementation of a logic (defined in section 5) consists of a specified class C of microkernel objects and a set O of operations for creating and extending objects in C using the microkernel operations. The microkernel is illustrated by a series of examples from the logic of the HOL theorem proving system [8]. Together, the examples indicate how the HOL logic can be implemented using the microkernel services.

In the future we plan to implement the microkernel and then use it to implement a logic based on a set theory called STMM [?].

This paper employs the following two notational conventions: Let $T = (A, B, \ldots, Z)$ be a tuple. Then (1) $A_T = A$, $B_T = B$, etc. and (2) $T_i = (A_i, B_i, \ldots, Z_i)$ for $i \in \{0, 1, 2, \ldots\}$.

2 The Language Service

The Language Service enables formal languages, language transformers, and assertions to be created as microkernel objects.

[3] There have been a number of research efforts to merge, in one way or another, computer algebra with computer theorem proving; see [1] for references.

2.1 Language Objects

A *language object* L is a tuple $(\mathcal{T}, \mathcal{N}, \mathcal{E}, \mathcal{C})$ such that:

1. $\mathcal{T}$ is a set of expressions called the *types* of L.
2. $\mathcal{N}$ is a nonempty set of symbols called the *names* of L.
3. $\mathcal{E}$ is a nonempty set of expressions called the *expressions* of L.
4. $\mathcal{C}$ is a nonempty set of functions called the *constructors* of L. If $\mathcal{T} \neq \emptyset$, each constructor is a partial function $c : \mathcal{N} \times \mathcal{T}^{<\omega} \times \mathcal{E}^{<\omega} \rightharpoonup \mathcal{E}$ such that, for all $n \in \mathcal{N}$, $\bar{\alpha} = \langle \alpha_1, \ldots, \alpha_m \rangle \in \mathcal{T}^{<\omega}$, and $\bar{e} = \langle e_1, \ldots, e_m \rangle \in \mathcal{E}^{<\omega}$, if $c(n, \bar{\alpha}, \bar{e})$ is defined, it is an expression
$$[\tilde{c}(n, \bar{\alpha}, \bar{e}) : \alpha]$$
where $\tilde{c}$ is a symbol that denotes the function c and $\alpha \in \mathcal{T}$. If $\mathcal{T} = \emptyset$, each constructor is a partial function $c : \mathcal{N} \times \mathcal{E}^{<\omega} \rightharpoonup \mathcal{E}$ such that, if $c(n, \bar{e})$ is defined, it is the expression $\tilde{c}(n, \bar{e})$.
5. $\mathcal{E}$ is the smallest set of expressions closed under the members of $\mathcal{C}$.

A language object L is intended to represent a formal typed language (when $\mathcal{T} \neq \emptyset$) or nontyped language (when $\mathcal{T} = \emptyset$). Every expression of L has a name (but a name such as no-name can be used for expressions intended to be unnamed). The *type* of an expression $e = [\tilde{c}(n, \bar{\alpha}, \bar{e}) : \alpha]$, written $\mathrm{tp}(e)$, is α. If L_1 and L_2 are language objects, then L_1 is a *sublanguage* of L_2, written $L_1 \leq L_2$, if $\mathcal{E}_1 \subseteq \mathcal{E}_2$.

Example 1. Let $\mathcal{N}_{tv}$ be a fixed infinite set of symbols called the *type variable names* of HOL. A *type structure* of HOL is a pair $\Omega = (\mathcal{N}_{tc}, a)$ such that:

1. $\mathcal{N}_{tc}$ is a set of symbols called the *type constant names* of Ω such that $\{*, \iota, \rightarrow\} \subseteq \mathcal{N}_{tc}$.
2. $a : \mathcal{N}_{tc} \rightarrow \mathbf{N}$ is a total function that maps each type constant name to its *arity*. $a(*) = a(\iota) = 0$ and $a(\rightarrow) = 2$.

A type structure $\Omega = (\mathcal{N}_{tc}, a)$ determines a language object $L_\Omega = (\emptyset, \mathcal{N}_t, \mathcal{E}, \mathcal{C})$ where $\mathcal{N}_t = \mathcal{N}_{tv} \cup \mathcal{N}_{tc}$ and $\mathcal{C}$ contains the constructors given in Table 1.

For convenience, we employ the following abbreviations:

n	for	$\mathsf{t\text{-}var}(n, \langle\rangle)$.
$(e_1, \ldots, e_m)n$	for	$\mathsf{t\text{-}const}(n, \langle e_1, \ldots, e_m \rangle)$.
$(e_1 \rightarrow e_2)$	for	$(e_1, e_2)\rightarrow$. □

c	$\tilde{c}$	c is defined on $(n, \bar{e}) \in \mathcal{N}_t \times \mathcal{E}^{<\omega}$
type-variable	t-var	iff $n \in \mathcal{N}_{tv}$ and $\bar{e} = \langle\rangle$
type-constant	t-const	iff $n \in \mathcal{N}_{tc}$ and $\lvert\bar{e}\rvert = a(n)$

Table 1. HOL Type Constructors

c	$\tilde{c}$	c is defined on $(n, \bar{\alpha}, \bar{e}) \in \mathcal{N} \times \mathcal{T}^{<\omega} \times \mathcal{E}^{<\omega}$	$\mathrm{tp}(c(n, \bar{\alpha}, \bar{e}))$
variable	var	only if $\bar{\alpha} = \langle\alpha\rangle$ and $\bar{e} = \langle\rangle$	α
constant	const	only if $\bar{\alpha} = \langle\alpha\rangle$ and $\bar{e} = \langle\rangle$	α
false	F	only if $\bar{\alpha} = \langle\rangle$ and $\bar{e} = \langle\rangle$	$*$
implication	$\supset$	only if $\bar{\alpha} = \langle\rangle$, $\bar{e} = \langle e_1, e_2\rangle$, and $\mathrm{tp}(e_1) = \mathrm{tp}(e_2) = *$	$*$
conjunction	$\wedge$	only if $\bar{\alpha} = \langle\rangle$ and the type of each member of $\bar{e}$ is $*$	$*$
equivalence	$\simeq$	only if $\bar{\alpha} = \langle\rangle$ and $\bar{e} = \langle e_1, e_2\rangle$	$*$
conditional	if	only if $\bar{\alpha} = \langle\rangle$, $\bar{e} = \langle e_1, e_2, e_3\rangle$, and $\mathrm{tp}(e_1) = *$	See note 1.

Notes:

1. If $c(n, \bar{\alpha}, \bar{e})$ is defined, its type will depend on the types of e_2 and e_3. For instance, if $\mathrm{tp}(e_2) = \mathrm{tp}(e_3) = \alpha$, then its type could be α or if $\mathrm{tp}(e_2)$ and $\mathrm{tp}(e_3)$ have a least upper bound β in a partial order on the types, then its type could be β.
2. Notice that the constructors in the table are not fully specified.

Table 2. Constructors of a Normal Language Object

A language object $L = (\mathcal{T}, \mathcal{N}, \mathcal{E}, \mathcal{C})$ is *normal* if:

1. There is a member of $\mathcal{T}$, denoted by $*$, that is intended to be the type of truth values.
2. no-name $\in \mathcal{N}$.
3. $\mathcal{C}$ includes the constructors given in Table 2.

For convenience, we employ the following abbreviations:

$\mathsf{var}(n, \alpha)$	for	$[\mathsf{var}(n, \langle\alpha\rangle, \langle\rangle) : \alpha]$.
$\mathsf{const}(n, \alpha)$	for	$[\mathsf{const}(n, \langle\alpha\rangle, \langle\rangle) : \alpha]$.
F	for	$[\mathsf{F}(\mathsf{no\text{-}name}, \langle\rangle, \langle\rangle) : *]$.
$(e_1 \supset e_2)$	for	$[\supset(\mathsf{no\text{-}name}, \langle\rangle, \langle e_1, e_2\rangle) : *]$.
$\neg e$	for	$(e \supset \mathsf{F})$.
T	for	$\neg\mathsf{F}$.
$\wedge(e_1, \ldots, e_n)$	for	$[\wedge(\mathsf{no\text{-}name}, \langle\rangle, \langle e_1, \ldots, e_n\rangle) : *]$.
$(e_1 \simeq e_2)$	for	$[\simeq (\mathsf{no\text{-}name}, \langle\rangle, \langle e_1, e_2\rangle) : *]$.
$\mathsf{if}(e_1, e_2, e_3, \alpha)$	for	$[\mathsf{if}(\mathsf{no\text{-}name}, \langle\rangle, \langle e_1, e_2, e_3\rangle) : \alpha]$.

Let L be a normal language object. A *formula* of L is an expression $e \in \mathcal{E}_L$ with $\mathrm{tp}(e) = *$. A *variable* and a *constant* of L is an expres-

c	$\tilde{c}$	c is defined on $(n, \bar{\alpha}, \bar{e}) \in \mathcal{N} \times \mathcal{T}^{<\omega} \times \mathcal{E}^{<\omega}$	$\mathrm{tp}(c(n, \bar{\alpha}, \bar{e}))$
variable	var	iff $n \in \mathcal{N}_v$, $\bar{\alpha} = \langle\alpha\rangle$, and $\bar{e} = \langle\rangle$	α
constant	const	iff $n \in \mathcal{N}_c$, $\bar{\alpha} = \langle\alpha\rangle$, $\mathrm{inst}(\alpha, t(n))$, and $\bar{e} = \langle\rangle$	α
application	@	iff $n = \textsf{no-name}$, $\bar{\alpha} = \langle\rangle$, $\bar{e} = \langle e_1, e_2\rangle$, and $\mathrm{tp}(e_1) = (\alpha \to \alpha')$ and $\mathrm{tp}(e_2) = \alpha$ for $\alpha, \alpha' \in \mathcal{T}$	$\mathrm{tp}(e_2)$
abstraction	λ	iff $n = \textsf{no-name}$, $\bar{\alpha} = \langle\rangle$, $\bar{e} = \langle e_1, e_2\rangle$, and e_1 is a variable	$\mathrm{tp}(e_1) \to \mathrm{tp}(e_2)$
false	F	iff $n = \textsf{no-name}$, $\bar{\alpha} = \langle\rangle$, and $\bar{e} = \langle\rangle$	$*$
implication	$\supset$	iff $n = \textsf{no-name}$, $\bar{\alpha} = \langle\rangle$, $\bar{e} = \langle e_1, e_2\rangle$, and $\mathrm{tp}(e_1) = \mathrm{tp}(e_2) = *$	$*$
conjunction	$\wedge$	iff $n = \textsf{no-name}$, $\bar{\alpha} = \langle\rangle$, and the type of each member of $\bar{e}$ is $*$	$*$
equivalence	$\simeq$	iff $n = \textsf{no-name}$, $\bar{\alpha} = \langle\rangle$, $\bar{e} = \langle e_1, e_2\rangle$, and $\mathrm{tp}(e_1) = \mathrm{tp}(e_2)$	$*$
conditional	if	iff $n = \textsf{no-name}$, $\bar{\alpha} = \langle\rangle$, $\bar{e} = \langle e_1, e_2, e_3\rangle$, $\mathrm{tp}(e_1) = *$, and $\mathrm{tp}(e_2) = \mathrm{tp}(e_3) = \alpha$	α

Notes:

1. $\mathrm{inst}(\alpha, t(n))$ means α is an instance of $t(n)$ obtained by substituting types for the type variables in $t(n)$.

Table 3. Constructors of an HOL Language Object

sion $e \in \mathcal{E}_L$ such that, for some $n \in \mathcal{N}_L$ and $\alpha \in \mathcal{T}_L$, $e = \textsf{var}(n, \alpha)$ and $e = \textsf{const}(n, \alpha)$, respectively. Variable binding constructors, such as quantifiers and the operator for lambda abstraction, can be represented by constructors that apply to variables.

Example 2. $\mathcal{N}_v$ be a fixed infinite set of symbols called the *variable names* of HOL such that $\textsf{no-name} \in \mathcal{N}_v$. A *signature* of HOL over a type structure Ω is a pair $\Sigma_\Omega = (\mathcal{N}_c, t)$ such that:

1. $\mathcal{N}_c$ is a set of symbols called the *constant names* of Σ_Ω such that $\mathcal{N}_c$ includes the names of constants of the HOL theory `LOG` [8, p. 215].
2. $t : \mathcal{N}_c \to \mathcal{E}_{L_\Omega}$ is a total function that maps each constant name to its *type*. t maps the names of the constants of `LOG` to their assigned types [8, p. 215].

The signature Σ_Ω determines an HOL *language object* $L_{\Sigma_\Omega} = (\mathcal{T}, \mathcal{N}, \mathcal{E}, \mathcal{C})$ where $\mathcal{T} = \mathcal{E}_{L_\Omega}$, $\mathcal{N} = \mathcal{N}_v \cup \mathcal{N}_c$, and $\mathcal{C}$ contains the constructors given in Table 3. It is easy to see that L_{Σ_Ω} is a normal language object.

For convenience, we employ the following abbreviations:

$e_1(e_2)$	for	$[@(\textsf{no-name}, \langle\rangle, \langle e_1, e_2\rangle) : \alpha]$.
$(\lambda\, e_1 \,.\, e_2)$	for	$[\lambda(\textsf{no-name}, \langle\rangle, \langle e_1, e_2\rangle) : \alpha]$. □

2.2 Transformer Objects

A *transformer object* $\mathit{\Pi}$ is a tuple (L_1, L_2, π) such that:

1. L_1 and L_2 are language objects called the *source* and *target languages* of $\mathit{\Pi}$, respectively.
2. π is a total function from $\mathcal{E}_1$ to $\mathcal{E}_2$ called the *transformer* of $\mathit{\Pi}$.

$\mathit{\Pi}$ *resides in* a language object L if $L_1, L_2 \leq L$.

A transformer object is intended to represent an expression transforming operation such as an evaluator, a simplifier, a rewrite rule, a rule of inference, a decision procedure, or an interpretation of one language in another. The notion of a transformer and machinery for defining transformers that are sound in an axiomatic theory are introduced in [6].

2.3 Assertion Objects

An *assertion object* A of a normal language L is a tuple (n, k, ξ) such that:

1. n is a symbol called the *name* of A.
2. $k \in \{0, 1, 2, 3\}$ denotes the *kind* of A.
3. If $k = 0$, then ξ is a formula of L.
4. If $k \in \{1, 2, 3\}$, then ξ is a transformer object residing in L.

An assertion object A is intended to be used either as an axiom that is assumed or as a theorem that is derived. An assertion object is *formulaic* if it is of kind 0. A formulaic assertion object $(n, 0, \varphi)$ is intended to assert that φ is valid. An assertion object is *transformational* if it is of kind 1, 2, or 3. A transformational assertion object $(n, k, \mathit{\Pi})$ where $\mathit{\Pi} = (L_1, L_2, \pi)$ is intended to assert that:

1. If $k = 1$, $\mathit{\Pi}$ is *computationally sound*, i.e., $e \simeq \pi(e)$ is valid for all expressions $e \in \mathcal{E}_1$.
2. If $k = 2$, $\mathit{\Pi}$ is *deductively sound*, i.e., $e \supset \pi(e)$ is valid for all formulas $e \in \mathcal{E}_1$.
3. If $k = 3$, $\mathit{\Pi}$ is *reductively sound*, i.e., $\pi(e) \supset e$ is valid for all formulas $e \in \mathcal{E}_1$.

2.4 Language Service Operations

The Language Service contains operations to create language, transformer, and assertion objects from their components, to create transformational assertion objects from transformer objects and other transformational assertion objects using transformer constructors (see [6]), and to apply a transformer object to an expression of a language object.

3 The Theory Service

The Theory Service offers objects for representing axiomatic theories, theorems, definitions, and interpretations between theories. The Theory Service does not provide operations for creating and extending Theory Service objects; these operations are provided by "logic implementations" which are defined in section 5. The motivation for the kinds of objects and their use in the Theory Service is discussed in [3].

3.1 Static Theory Objects

A *static theory object* T is a tuple $(n, L, \mathit{\Gamma}, J)$ such that:

1. n is the name of a logic implementation (defined in section 5) called the *logic* of T.
2. L is a normal language object called the *language* of L.
3. $\mathit{\Gamma}$ is a set of assertion objects of L called the *axioms* of L.
4. J is an unspecified object called the *justification* of T.

A static theory object T is intended to represent an axiomatic theory whose axioms are presented by assertion objects. The justification of T is intended to prove that T represents an axiomatic theory of the logic of T. If T_1 and T_2 are static theory objects, then T_1 is a *subtheory* of T_2, written $T_1 \leq T_2$, if $n_1 = n_2$, $L_1 \leq L_2$, and $\mathit{\Gamma}_1 \subseteq \mathit{\Gamma}_2$.

Example 3. A *static theory object* of HOL is a static theory object $(\mathsf{hol}, L, \mathit{\Gamma}, J)$ such that $L = L_{\mathit{\Sigma}_\mathit{\Omega}}$ where $\mathit{\Sigma}_\mathit{\Omega}$ is a signature of HOL, J is empty, and $\mathit{\Gamma}$ includes the following assertions:

1. For each axiom φ of the HOL theory `INIT` [8, p. 218], an assertion object of L of kind 0 that represents φ.
2. For each rule of inference R of the HOL logic [8, pp. 212–213], an assertion object of L of kind 2 that represents R as a deductively sound transformer. A sequent $\{t_1, \ldots, t_n\} \vdash t$ in the HOL logic is represented by a formula $\wedge(\hat{t}_1, \ldots, \hat{t}_n) \supset \hat{t}$ of L. $\Box$

3.2 Theorem Objects

A *theorem object* H is a tuple (T, A, J) such that:

1. T is a static theory object.
2. A is an assertion object of L_T.
3. J is an unspecified object called the *justification* of H.

The *logic* of H is the logic of T.

A theorem object H is intended to assert a formula or transformer in the axiomatic theory represented by T. The justification of H is intended to prove that the assertion represented by A holds in the axiomatic theory represented by T.

Example 4. A *theorem object* of HOL is theorem object (T, A, J) such that T is a static theory object of HOL, $A = (n, 0, \varphi)$ is an assertion object of L_T , and J is a proof that φ is valid in T.

3.3 Definition Objects

A *definition object* D is a tuple (T, c, e, J) such that:

1. T is a static theory object.
2. c is a constant not in L_T.
3. e is an expression of L_T.
4. J is an unspecified object called the *justification* of D.

The *logic* of D is the logic of T.

A definition object D is intended to assert that the new constant c is equivalent to the expression e. The justification of D is intended to prove that the addition of the formula $c \simeq e$ to the axiomatic theory represented by T is conservative.

Example 5. A *definition object* of HOL is a definition object (T, c, e, J) such that T is a static theory object of HOL, e contains no free variables and all the type variables occurring in e also occur in $\mathrm{tp}(c)$, and J is empty. A definition object of HOL represents a *constant definition* of the HOL logic [8, p. 220].

3.4 Profile Objects

A *profile object* P is a tuple (T, e, J) such that:

1. T is a static theory object.
2. e is a formula not in L_T.
3. J is an unspecified object called the *justification* of P.

The *logic* of P is the logic of T.

A profile object P is intended to assert that a set of new types and expressions satisfies the properties specified by the formula e. The justification of P is intended to prove that the addition of the formula e to the axiomatic theory represented by T is conservative. A profile object is a generalization of a definition object that can be used to introduce new types and expressions and new type and expression constructors and to extend the domains of old type and expression constructors.

Example 6. A *profile object* of HOL is a profile object that represents a *constant specification*, a *type definition*, or a *type specification* of the HOL logic [8, pp. 222–232]. The details are left to the reader.

3.5 Dynamic Theory Objects

A *dynamic theory object* U is a tuple $(n, T, \sigma, \mathcal{N}, J)$ such that:

1. n is a symbol called the *name* of U.
2. T is static theory object called the *base theory* of U.
3. σ is a finite sequence of theorem, definition, and profile objects called the *event history* of T. The logic of each member of σ is the logic of T.
4. $\mathcal{N}$ is a finite set of names of other dynamic theory objects called the *principal subtheories* of U. For each principal subtheory U' of U, the logic of U' is the logic of T, the base theory of U' is a subtheory of T, and the event history of U' is a subsequence of σ.
5. J is an unspecified object called the *justification* of U.

The *logic* of U is the logic of T. If τ is an initial segment of σ, then τ determines a static theory denoted as $T\tau$ such that $T \leq T\tau$. The static theory object $T\sigma$ is called the *current theory* of U.

A dynamic theory object U is intended to represent an axiomatic theory created by extending a base axiomatic theory with theorems, definitions, and profiles. The justification of U is intended to prove that the axiomatic theory represented by the current theory of U is a conservative extension of the axiomatic theory represented by the base theory of U.

3.6 Static Interpretation Objects

A *static interpretation object* Φ is a tuple (T_1, T_2, Π, J) such that:

1. $T_i = (n_i, L_i, \Gamma_i, J_i)$ is a static theory object for $i = 1, 2$. T_1 and T_2 are called the *source* and *target theories* of Φ, respectively.
2. $\Pi = (L'_1, L'_2, \pi)$ is a transformer object such that $L_i \leq L'_i$ for $i = 1, 2$ called the *interpretation* of Φ.
3. J is an unspecified object called the *justification* of Φ.

The *logic* of Φ is the logic of T_2. Φ is an *intralogic* interpretation if the logics of T_1 and T_2 are the same and is an *interlogic* interpretation if the logics of T_1 and T_2 are different.

A static interpretation Φ is intended to represent a interpretation from one axiomatic theory in another. The justification of Φ is intended to prove that the interpretation maps formulas valid in the source theory to formulas valid in the target theory.

Example 7. Interpretations of one theory in another are not part of the HOL logic. However, a "homomorphic" notion of an HOL interpretation could be defined by lifting the standard first-order notion of an interpretation to the HOL logic. However, it would be problematic for an HOL interpretation to associate a type in the source theory with a proper subtype in the target theory as first-order interpretations do because such an association would lead to partial functions, which are not directly supported in the HOL logic (see [2] for examples and further discussion).

3.7 Dynamic Interpretation Objects

A *dynamic interpretation object* Ψ is a tuple $(n, U_1, U_2, \tau_1, \tau_2, \Pi, J)$ such that:

1. n is a symbol called the *name* of Ψ.
2. $U_i = (n_i, T_i, \sigma_i, \mathcal{N}_i, J_i)$ is a dynamic theory object for $i = 1, 2$. U_1 and U_2 are called the *source* and *target theories* of Ψ, respectively.
3. τ_i is an initial segment of σ_i for $i = 1, 2$.
4. $(T_1\tau_1, T_2\tau_2, \Pi, J)$ is a static interpretation object called the *current interpretation* of Ψ.

The *logic* of Ψ is the logic of U_2.

A dynamic interpretation object Ψ is intended to represent an interpretation of a conservative extension of one axiomatic theory in a conservative extension of another axiomatic theory.

4 The Derivation Service

The Derivation Service enables computations and deductions to be represented that use the machinery of dynamic theory objects. The motivation and use of the Derivation Service is discussed in [7].

Name	Tuple	Conditions
implication	$(1, N_1, N_2)$	N_1, N_2 are formulaic.
negation	$(2, N_1, N_2)$	N_1, N_2 are formulaic.
one-to-many	$(3, N, \{N_1, \ldots, N_m\})$	$N, N_1, \ldots, N_m$ are formulaic.
equivalence	$(4, N_1, N_2)$	
conditional equivalence	$(5, N, N_1, N_2)$	N is formulaic.
free	$(6, N_1, N_2)$	

Table 4. Connector Objects

4.1 Node Objects

A *node object* N is a tuple (U, e) such that:

1. U is dynamic theory object.
2. e is an expression of the language of the current theory of U.

The *logic* of N is the logic of U. N is *formulaic* if e is a formula.

A node object N is intended to represent an expression in the context of the axiomatic theory represented by the current theory of U.

4.2 Connector Objects

The six kinds of *connector objects* are given in Table 4. Each connector object is a tuple consisting of a *kind* $k \in \{1, \ldots, 6\}$ and a collection of node objects.

A connector object C is intended to record that its component node objects are related in a certain way. An implication connector object records that N_1 logically implies N_2. A negation connector object records that N_2 is the logical negation of N_1. A one-to-many connector object records that N is logically equivalent to the conjunction of $N_1, \ldots, N_m$. An equivalence connector object records that N_1 and N_2 have the same value. A conditional equivalence connector object records that N_1 and N_2 have the same value provided N holds. A free connector object records that N_1 and N_2 are related in some unspecified way.

4.3 Derivation Graph Objects

A *derivation graph object* G is a tuple $(n, \mathcal{N}, \mathcal{C})$ such that:

1. n is the name of a logic implementation (defined in section 5) called the *logic* of G.

2. $\mathcal{N}$ is a finite set of node objects N such that the logic of N is the logic of G.
3. $\mathcal{C}$ is a finite set of connector objects that contain no node objects outside of $\mathcal{N}$.

A derivation graph object is intended to record a web of computations and deductions. Sequences of node objects connected by (conditional) equivalence connectors represent (conditional) computations, while trees of formulaic node objects connected by implication, negation, one-to-many, and equivalence connectors represent deductions.

4.4 Derivation Service Operations

The Derivation Service provides operations for creating each kind of Derivation Service object and five *derivation graph operations* for adding node objects and connector objects to derivation graph objects.

If U is a dynamic theory object whose current theory is (n, L, Γ, J) and e is a formula of L, then $U[e]$ is some dynamic theory object whose current theory is $(n', L, \Gamma \cup \{(n'', 0, e)\}, J')$ for some symbols n' and n''. $U[e]$ represents the result of adding the formula e to U as a new axiom.

The five derivation graph operations are defined in Table 5. Each operation takes a derivation graph object $G = (n, \mathcal{N}, \mathcal{C})$ and other objects (the input objects) and returns a derivation graph object

$$G' = (n, \mathcal{N} \cup \mathcal{N}', \mathcal{C} \cup \mathcal{C}')$$

obtained by adding a finite set $\mathcal{N}'$ of node objects (the output node objects) and a finite set $\mathcal{C}'$ of connector objects (the output connector objects) to G.

Remark 1. In a logic implementation, a derivation graph object G can be modified both by (1) applying derivation graph operations to G and (2) applying operations of the logic implementation that extend the dynamic theory objects contained in G.

5 Logic Implementations

A *logic implementation* Λ is a tuple (n, S, O) such that:

1. n is a symbol called the *name* of Λ.

Name	Input Objects	Output Objects
add-node	$N = (U, e)$	N
apply-transformer	k $N = (U, e) \in \mathcal{N}$ $\Pi = (L_1, L_2, \pi)$	$N' = (U, \pi(e))$ $C = (6, N, N')$ if $k = 0$ $C = (4, N, N')$ if $k = 1$ $C = (1, N, N')$ if $k = 2$ $C = (1, N', N)$ if $k = 3$
split-implication	$N = (U, e_1 \supset e_2) \in \mathcal{N}$	$N' = (U[e_1], e_2)$ $C = (4, N, N')$
split-conjunction	$N = (U, \wedge(e_1, \ldots, e_m)) \in \mathcal{N}$	$N_i = (U, e_i)$ for $i = 1, \ldots, m$ $C = (3, N, \{N_1, \ldots, N_m\})$
split-condition	$N = (U, \mathsf{if}(e_1, e_2, e_3, \alpha)) \in \mathcal{N}$	$N_1 = (U, e_1)$ $N_1' = (U, \neg e_1)$ $N_2 = (U[e_1], e_2)$ $N_3 = (U[\neg e_1], e_3)$ $C_1 = (5, N_1, N, N_2)$ $C_2 = (5, N_1', N, N_3)$ $C_3 = (2, N_1, N_1')$

Notes:

1. Π is a transformer residing in the language of the current theory of U.
2. k = 1, 2, or 3 means that Π is computationally sound, deductively sound, or reductively sound, respectively, in the current theory of U. $k = 0$ means that nothing is assumed about the soundness of Π in the current theory of U.

Table 5. The Derivation Graph Operations

2. S is a specification of the class of Theory Service objects that belong to Λ. Λ is the logic of each Theory Service object belonging to Λ. S may require that the Theory Service objects belonging to Λ include certain specified justifications.
3. O is a set of operations for creating and extending microkernel objects using the microkernel operations. For example, O could include operations for:

   (a) Creating a Theory Service object belonging to Λ from its components using the Language Service operations.
   (b) Creating a formulaic assertion object belonging to Λ from a derivation graph (which proves the formula of the assertion object) belonging to Λ.
   (c) Creating a transformational assertion object belonging to Λ from a formulaic assertion object belonging to Λ in the style of IMPS macetes [5] (see [6] for details and examples).

(d) Extending a dynamic theory object belonging to Λ by adding theorem, definition, and profile objects belonging to Λ to its event history.
(e) Extending a dynamic interpretation object belonging to Λ by extending its current interpretation.

Example 8. A standard implementation of HOL using the microkernel would be a logic implementation (hol, S, O) such that:

1. S says that a Theory Service object belongs to the implementation of HOL iff it is a Theory Service object of HOL as illustrated in the examples of section 3.
2. O contains the kinds of operations listed above in (a), (b), and (d).

References

1. M. N. Dunstan, T. Kelsey, U. Martin, and S. Linton. Formal methods for extensions to CAS. In J.M. Wing, J. Woodcock, and J. Davies, editors, *FM'99—Formal Methods, Volume II*, volume 1709 of *Lecture Notes in Computer Science*, pages 1758–1777. Springer-Verlag, 1999.
2. W. M. Farmer. Theory interpretation in simple type theory. In J. Heering et al., editor, *Higher-Order Algebra, Logic, and Term Rewriting*, volume 816 of *Lecture Notes in Computer Science*, pages 96–123. Springer-Verlag, 1994.
3. W. M. Farmer. An infrastructure for intertheory reasoning. In D. McAllester, editor, *Automated Deduction—CADE-17*, volume 1831 of *Lecture Notes in Computer Science*, pages 115–131. Springer-Verlag, 2000.
4. W. M. Farmer, J. D. Guttman, and F. J. Thayer Fábrega. IMPS: An updated system description. In M. McRobbie and J. Slaney, editors, *Automated Deduction—CADE-13*, volume 1104 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, 1996.
5. W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11:213–248, 1993.
6. W. M. Farmer and M. v. Mohrenschildt. Transformers for symbolic computation and formal deduction. In S. Colton, U. Martin, and V. Sorge, editors, *CADE-17 Workshop on the Role of Automated Deduction in Mathematics*, pages 36–45, 2000.
7. W. M. Farmer and M. v. Mohrenschildt. A unified framework for computer algebra and theorem proving systems. 2001.
8. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.
9. R. Nickson, O. Traynor, and M. Utting. Cogito ergo sum—providing structured theorem prover support for specification formalisms. In K. Ramamohanarao, editor, *Proceedings of the Nineteenth Australasian Computer Science Conference*, volume 18 of *Australian Computer Science Communications*, pages 149–158, 1997.
10. L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science.* Springer-Verlag, 1994.