An Infrastructure for Intertheory Reasoning*

William M. Farmer

Department of Computing and Software
McMaster University
1280 Main Street West
Hamilton, Ontario L8S 4L7
Canada

wmfarmer@mcmaster.ca

2 May 2000

Abstract. The little theories method, in which mathematical reasoning is distributed across a network of theories, is a powerful technique for describing and analyzing complex systems. This paper presents an infrastructure for intertheory reasoning that can support applications of the little theories method. The infrastructure includes machinery to store theories and theory interpretations, to store known theorems of a theory with the theory, and to make definitions in a theory by extending the theory "in place". The infrastructure is an extension of the intertheory infrastructure employed in the IMPS Interactive Mathematical Proof System.

1 Introduction

Mathematical reasoning is always performed within some context, which includes vocabulary and notation for expressing concepts and assertions, and axioms and inference rules for proving conjectures. In informal mathematical reasoning, the context is almost entirely implicit. In fact, substantial mathematical training is often needed to "see" the context.

The situation is quite different in formal mathematics performed in logical systems often with the aid of computers. The context is formalized as a mathematical structure. The favored mathematical structure for this purpose is an axiomatic theory within a formal logic. An axiomatic theory, or *theory* for short, consists of a formal language and a set of axioms expressed in the language. It is a specification of a set of objects: the language provides names for the objects and the axioms constrain what properties the objects have.

Sophisticated mathematical reasoning usually involves several related but different mathematical contexts. There are two main ways of dealing with a

* Presented at the 17th International Conference on Automated Deduction (CADE-17), Pittsburgh, PA, USA, June 2000. Published in: D. McAllester, ed., *Automated Deduction—CADE-17*, *Lecture Notes in Computer Science*, Vol. 1831, pp. 115–131, Springer-Verlag, 2000.

multitude of contexts using theories. The *big theory method* is to choose a highly expressive theory—often based on set theory or type theory—that can represent many different contexts. Each context that arises is represented in the theory or in an extension of the theory. Contexts are related to each other in the theory itself.

An alternate approach is the *little theories method* in which separate contexts are represented by separate theories. Structural relationships between contexts are represented as interpretations between theories (see [4, 19]). Interpretations serve as conduits for passing information (e.g., definitions and theorems) from abstract theories to more concrete theories, or indeed to other equally abstract theories. As a result, the big theory is replaced with a network of theories—which can include both small compact theories and large powerful theories. The little theories approach has been used in both mathematics and computer science (see [10] for references). In [10] we argue that the little theories method offers important advantages for mechanized mathematics. Many of these advantages have been demonstrated by the IMPS Interactive Mathematical Proof System [9, 11] which supports the little theories method.

A mechanized mathematics system based on the little theories method requires a different infrastructure than one based on the big theory method. In the big theory method all reasoning is performed within a single theory, while in the little theories method there is both intertheory and intratheory reasoning. This paper presents an infrastructure for intertheory reasoning that can be employed in several kinds of mechanized mathematics systems including theorem provers, software specification and verification systems, computer algebra systems, and electronic mathematics libraries. The infrastructure is closely related to the intertheory infrastructure used in IMPS, but it includes some capabilities which are not provided by the IMPS intertheory infrastructure.

The little theories method is a major element in the design of several software specification systems including EHDM [18], IOTA [16], KIDS [20], OBJ3 [12], and Specware [21]. The intertheory infrastructures of these systems are mainly for constructing theories and linking them together into a network. They do not support the rich interplay of making definitions, proving theorems, and "transporting" definitions and theorems from one theory to another needed for developing and exploring theories within a network.

The Ergo [17] theorem proving system is another theorem proving system besides IMPS that directly supports the little theories method.[1] Its infrastructure for intertheory reasoning provides full support for *constructing theories* from other theories via inclusion and interpretation but only partial support for *developing theories* by making definitions and proving theorems. In Ergo, theory interpretation is static: theorems from the source theory of an interpretation can be transported to the target theory of the interpretation only when the interpretation is created [14]. Theory interpretation is dynamic in the intertheory infrastructure of this paper (and of IMPS).

[1] Many theorem proving systems indirectly support the little theories methods by allowing a network of theories to be formalized within a big theory.

The rest of the paper is organized as follows. The underlying logic of the intertheory infrastructure is given in section 2. Section 3 discusses the design requirements for the infrastructure. The infrastructure itself is presented in section 4. Finally, some applications of the infrastructure are described in section 5.

2 The Underlying Logic

The intertheory infrastructure presented in this paper assumes an underlying logic. Many formal systems, including first-order logic and Zermelo-Fraenkel set theory, could serve as the underlying logic. For the sake of convenience and precision, we have chosen a specific underlying logic for the infrastructure rather than treating the underlying logic as a parameter. Our choice is Church's simple theory of types [3], denoted in this paper by **C**.

The underlying logics of many theorem proving systems are based on **C**. For example, the underlying logic of IMPS (and its intertheory infrastructure) is a version of **C** called LUTINS [5, 6, 8]. Unlike **C**, LUTINS admits undefined terms, partial functions, and subtypes. By virtue of its support for partial functions and subtypes, many theory interpretations can be expressed more directly in LUTINS than in **C** [8].

We will give now a brief presentation of **C**. The missing details can be filled in by consulting Church's original paper [3] or one of the logic textbooks, such as [1], which contains a full presentation of **C**. We will also define a number of logical notions in the context of **C** including the notions of a theory and an interpretation.

2.1 Syntax of C

The *types* of **C** are defined inductively as follows:

1. ι is a type (which denotes the type of individuals).
2. $*$ is a type (which denotes the type of truth values).
3. If α and β are types, then $(\alpha \to \beta)$ is a type (which denotes the type of total functions that map values of type α to values of type β).

Let $\mathcal{T}$ denote the set of types of **C**.

A *tagged symbol* is a symbol tagged with a member of $\mathcal{T}$. A tagged symbol whose symbol is a and whose tag is α is written as a_α. Let $\mathcal{V}$ be a set of tagged symbols called *variables* such that, for each $\alpha \in \mathcal{T}$, the set of members of $\mathcal{V}$ tagged with α is countably infinite. A *constant* is a tagged symbol c_α such that $c_\alpha \notin \mathcal{V}$.

A *language* L of **C** is a set of constants. (In the following, let a "language" mean a "language of **C**".) An *expression of type* α of L is a finite sequence of symbols defined inductively as follows:

1. Each $a_\alpha \in \mathcal{V} \cup L$ is an expression of type α.
2. If F is an expression of type $\alpha \to \beta$ and A is an expression of type α, then $F(A)$ is an expression of type β.

3. If $x_\alpha \in \mathcal{V}$ and E is an expression of type β, then $(\lambda\, x_\alpha\, .\, E)$ is an expression of type $\alpha \to \beta$.
4. If A and B are expressions of type α, then $(A = B)$ is an expression of type $*$.
5. If A and B are expressions of type $*$, then $\neg A$, $(A \supset B)$, $(A \wedge B)$, and $(A \vee B)$ are expressions of type $*$.
6. If $x_\alpha \in \mathcal{V}$ and E is an expression of type $*$, then $(\forall\, x_\alpha\, .\, E)$ and $(\exists\, x_\alpha\, .\, E)$ are expressions of type $*$.

Expressions of type α are denoted by $A_\alpha, B_\alpha, C_\alpha$, etc. Let $\mathcal{E}_L$ denote the set of expressions of L. "Free variable", "closed expression", and similar notions are defined in the obvious way. Let $\mathcal{S}_L$ denote the set of *sentences* of L, i.e., the set of closed expressions of type $*$ of L.

2.2 Semantics of C

For each language L, there is a set $\mathcal{M}_L$ of *models* and a relation $\models$ between models and sentences of L. $M \models A_*$ is read as "M is a model of A_*". Let L be a language, $A_* \in \mathcal{S}_L$, $\Gamma \subseteq \mathcal{S}_L$, and $M \in \mathcal{M}_L$. M is a *model of* Γ, written $M \models \Gamma$, if $M \models B_*$ for all $B_* \in \Gamma$. Γ *logically implies* A_*, written $\Gamma \models A_*$, if every model of Γ is a model of A_*.

2.3 Theories

A *theory* of **C** is a pair $T = (L, \Gamma)$ where L is a language and $\Gamma \subseteq \mathcal{S}_L$. Γ serves as the set of axioms of T. (In the following, let a "theory" mean a "theory of **C**".) A_* is a *(semantic) theorem* of T, written $T \models A_*$, if $\Gamma \models A_*$. T is *consistent* if some sentence of L is not a theorem of T. A theory $T' = (L', \Gamma')$ is an *extension* of T, written $T \leq T'$, if $L \subseteq L'$ and $\Gamma \subseteq \Gamma'$. T' is a *conservative extension* of T, written $T \trianglelefteq T'$, if $T \leq T'$ and, for all $A_* \in \mathcal{S}_L$, if $T' \models A_*$, then $T \models A_*$.

The following lemma about theory extensions is easy to prove.

Lemma 1. *Let T_1, T_2, and T_3 be theories.*

1. *If $T_1 \leq T_2 \leq T_3$, then $T_1 \leq T_3$.*
2. *If $T_1 \trianglelefteq T_2 \trianglelefteq T_3$, then $T_1 \trianglelefteq T_3$.*
3. *If $T_1 \leq T_2 \leq T_3$ and $T_1 \trianglelefteq T_3$, then $T_1 \trianglelefteq T_2$.*
4. *If $T_1 \trianglelefteq T_2$ and T_1 is consistent, then T_2 is consistent.*

2.4 Interpretations

Let $T = (L, \Gamma)$ and $T' = (L', \Gamma')$ be theories, and let $\Phi = (\gamma, \mu, \nu)$ where $\gamma \in \mathcal{T}$ and $\mu : \mathcal{V} \to \mathcal{V}$ and $\nu : L \to \mathcal{E}_{L'}$ are total functions.

For $\alpha \in \mathcal{T}$, $\Phi(\alpha)$ is defined inductively as follows:

1. $\Phi(\iota) = \gamma$.
2. $\Phi(*) = *$.
3. If $\alpha, \beta \in \mathcal{T}$, then $\Phi(\alpha \to \beta) = \Phi(\alpha) \to \Phi(\beta)$.

Φ is a *translation from L to L'* if:

1. For all $x_\alpha \in \mathcal{V}$, $\mu(x_\alpha)$ is of type $\Phi(\alpha)$.
2. For all $c_\alpha \in L$, $\nu(c_\alpha)$ is of type $\Phi(\alpha)$.

Suppose Φ is a translation from L to L'. For $E_\alpha \in \mathcal{E}_L$, $\Phi(E_\alpha)$ is the member of $\mathcal{E}_{L'}$ defined inductively as follows:

1. If $E_\alpha \in \mathcal{V}$, then $\Phi(E_\alpha) = \mu(E_\alpha)$.
2. If $E_\alpha \in L$, then $\Phi(E_\alpha) = \nu(E_\alpha)$.
3. $\Phi(F_{\alpha\to\beta}(A_\alpha)) = \Phi(F_{\alpha\to\beta})(\Phi(A_\alpha))$.
4. $\Phi(\lambda\, x_\alpha\,.\, E_\beta) = (\lambda\, \Phi(x_\alpha)\,.\, \Phi(E_\beta))$.
5. $\Phi(A_\alpha = B_\alpha) = (\Phi(A_\alpha) = \Phi(B_\alpha))$
6. $\Phi(\neg E_*) = \neg\Phi(E_*)$.
7. $\Phi(A_* \,\square\, B_*) = (\Phi(A_*) \,\square\, \Phi(B_*))$ where $\square \in \{\supset, \wedge, \vee\}$.
8. $\Phi(\square x_\alpha\,.\, E_*) = (\square\Phi(x_\alpha)\,.\, \Phi(E_*))$ where $\square \in \{\forall, \exists\}$.

Φ is an *interpretation of T in T'* if it is a translation from L to L' that maps theorems to theorems, i.e., for all $A_* \in \mathcal{S}_L$, if $T \models A_*$, then $T' \models \Phi(A_*)$.

Theorem 1 (Relative Consistency). *Suppose Φ be an interpretation of T in T' and T' is consistent. Then T is consistent.*

Proof. Assume $\Phi = (\gamma, \mu, \nu)$ is an interpretation of T in T', T' is consistent, and T is inconsistent. Then $F_* = (\exists\, x_\iota\,.\, \neg(x_\iota = x_\iota))$ is a theorem of T, and so $\Phi(F_*) = (\exists\, \mu(x_\iota)\,.\, \neg(\mu(x_\iota) = \mu(x_\iota)))$ is a theorem of T', which contradicts the consistency of T'.□

The next theorem gives a sufficient condition for a translation to be an interpretation.

Theorem 2 (Interpretation Theorem). *Suppose Φ is a translation from L to L' and, for all $A_* \in \Gamma$, $T' \models \Phi(A_*)$. Then Φ is an interpretation of T in T'.*

Proof. The proof is similar to the proof of Theorem 12.4 in [6].□

3 Design Requirements

At a minimum, an infrastructure for intertheory reasoning should provide the capabilities to store theories and interpretations and to record theorems as they are discovered. We present in this section a "naive" intertheory infrastructure with just these capabilities. We then show that the naive infrastructure lacks several important capabilities. From these results we formulate the requirements that an intertheory infrastructure should satisfy.

3.1 A Naive Intertheory Infrastructure

We present now a naive intertheory infrastructure. In this design, the state of the infrastructure is a set of *infrastructure objects*. The infrastructure state is initially the empty set. It is changed by the application of *infrastructure operations* which add new objects to the state or modify objects already in the state. There are three kinds of infrastructure objects for storing theories, theorems, and interpretations, respectively, and there are four infrastructure operations for creating the three kinds of objects and for "installing" theorems in theories.

Infrastructure objects are denoted by boldface letters. The three infrastructure objects are defined simultaneously as follows:

1. A *theory object* is a tuple $\mathbf{T} = (n, L, \Gamma, \Sigma)$ where n is a string, L is a language, $\Gamma \subseteq \mathcal{S}_L$, and Σ is a set of theorem objects. n is called the *name* of $\mathbf{T}$ and is denoted by $[\mathbf{T}]$. (L, Γ) is called the *theory* of $\mathbf{T}$ and is denoted by $\mathsf{thy}(\mathbf{T})$.
2. A *theorem object* is a tuple $\mathbf{A} = ([\mathbf{T}], A_*, J)$ where $\mathbf{T} = (n, L, \Gamma, \Sigma)$ is a theory object, $A_* \in \mathcal{S}_L$, and J is a justification[2] that $\mathsf{thy}(\mathbf{T}) \models A_*$.
3. An *interpretation object* is a tuple $\mathbf{I} = ([\mathbf{T}], [\mathbf{T}'], \Phi, J)$ where $\mathbf{T}$ and $\mathbf{T}'$ are theory objects, Φ is a translation, and J is a justification that Φ is an interpretation of $\mathsf{thy}(\mathbf{T})$ in $\mathsf{thy}(\mathbf{T}')$.

Let $\mathbf{S}$ denote the infrastructure state. The four infrastructure operations are defined as follows:

1. Given a string n, a language L, and $\Gamma \subseteq \mathcal{S}_L$ as input, if, for all theory objects $\mathbf{T}' = (n', L', \Gamma', \Sigma') \in \mathbf{S}$, $n \neq n'$ and $\mathsf{thy}(\mathbf{T}) \neq \mathsf{thy}(\mathbf{T}')$, then create-thy-obj adds the theory object $(n, L, \Gamma, \emptyset)$ to $\mathbf{S}$; otherwise, the operation fails.
2. Given a theory object $\mathbf{T} \in \mathbf{S}$, a sentence A_*, and a justification J as input, if $\mathbf{A} = ([\mathbf{T}], A_*, J)$ is a theorem object, then create-thm-obj adds $\mathbf{A}$ to $\mathbf{S}$; otherwise, the operation fails.
3. Given two theory objects $\mathbf{T}, \mathbf{T}' \in \mathbf{S}$, a translation Φ, and a justification J as input, if $\mathbf{I} = ([\mathbf{T}], [\mathbf{T}'], \Phi, J)$ is an interpretation object, then create-int-obj adds $\mathbf{I}$ to $\mathbf{S}$; otherwise, the operation fails.
4. Given a theorem object $\mathbf{A} = ([\mathbf{T}], A_*, J) \in \mathbf{S}$ and a theory object $\mathbf{T}' = (n', L', \Gamma', \Sigma') \in \mathbf{S}$ as input, if $\mathsf{thy}(\mathbf{T}) \leq \mathsf{thy}(\mathbf{T}')$, then install-thm-obj replaces $\mathbf{T}'$ in $\mathbf{S}$ with the theory object $(n', L', \Gamma', \Sigma' \cup \{\mathbf{A}\})$; otherwise, the operation fails.

[2] The notion of a justification is not specified. It could, for example, be a formal proof.

3.2 Missing Capabilities

The naive infrastructure is missing four important capabilities:

A. Definitions Suppose we would like to make a definition that the constant $\mathsf{is\_zero}_{\iota\to *}$ is the predicate $(\lambda\, x_\iota\, .\, x_\iota = 0_\iota)$ in a theory T stored in a theory object $\mathbf{T} = (n, L, \Gamma, \Sigma) \in \mathbf{S}$. The naive infrastructure offers only one way to do this: create the extension $T' = (L', \Gamma')$ of T, where $L' = L \cup \{\mathsf{is\_zero}_{\iota\to *}\}$ and

$$\Gamma' = \Gamma \cup \{\mathsf{is\_zero}_{\iota\to *} = (\lambda\, x_\iota\, .\, x_\iota = 0_\iota)\},$$

and then store T' in a new theory object $\mathbf{T}'$ by invoking create-thy-obj. If $\mathsf{is\_zero}_{\iota\to *}$ is not in L, T and T' can be regarded as the same theory since $T \trianglelefteq T'$ and $\mathsf{is\_zero}_{\iota\to *}$ can be "eliminated" from any expression of L' by replacing every occurrence of it with $(\lambda\, x_\iota\, .\, x_\iota = 0_\iota)$.

Definitions are made all the time in mathematics, and thus, implementing definitions in this way will lead to an explosion of theory objects storing theories that are essentially the same. A better way of implementing definitions would be to extend T to T' "in place" by replacing T in $\mathbf{T}$ with T'. The resulting object would still be a theory object because every theorem of T is also a theorem of T'.

This approach, however, would introduce a new problem. If an interpretation object $\mathbf{I} = ([\mathbf{T}], [\mathbf{T}'], \Phi, J) \in \mathbf{S}$ and $\mathsf{thy}(\mathbf{T})$ is extended in place by making a definition $c_\alpha = E_\alpha$, then the interpretation Φ would no longer be an interpretation of T in T' because $\Phi(c_\alpha)$ would not be defined.

There are three basic solutions to this problem. The first one is to automatically extend Φ to an interpretation of T in T' by defining $\Phi(c_\alpha) = \Phi(E_\alpha)$. However, this solution has the disadvantage that, when an expression of T containing c_α is translated to an expression of T' via the extended Φ, the expression of T will be expanded into a possibly much bigger expression of T'.

The second solution is to automatically transport the definition $c_\alpha = E_\alpha$ from T to a T' via Φ by making a new definition of the form $d_\beta = \Phi(E_\alpha)$ in T' and defining $\Phi(c_\alpha) = d_\beta$. The implementation of this solution would require care because, when two similar theories are both interpreted in a third theory, common definitions in the source theories may be transported multiple times to the target theory, resulting in definitions in the target theory that define different constants in exactly the same way.

The final solution is to let the user extend Φ by hand whenever it is necessary. This solution is more flexible than the first two solutions, but it would impose a heavy burden on the user. Our experience in developing IMPS suggests that the best solution would be some combination of these three basic solutions.

B. Profiles Suppose we would like to make a "definition" that the constant $\mathsf{a\_non\_zero}_\iota$ has a value not equal to 0_ι in a theory T stored in a theory object $\mathbf{T} = (n, L, \Gamma, \Sigma) \in \mathbf{S}$. That is, we would like to add a new constant $\mathsf{a\_non\_zero}_\iota$ to L whose value is specified, but not necessarily uniquely determined, by the

sentence $\neg(\mathsf{a\_non\_zero}_\iota = 0_\iota)$. More precisely, let $T' = (L', \Gamma')$ where $L' = L \cup \{\mathsf{a\_non\_zero}_\iota\}$ and

$$\Gamma' = \Gamma \cup \{\neg(\mathsf{a\_non\_zero}_\iota = 0_\iota)\}.$$

If $\mathsf{a\_non\_zero}_\iota$ is not in L and the sentence $(\exists\, x_\iota \,.\, \neg(x_\iota = 0_\iota))$ is a theorem of T, then $T \trianglelefteq T'$.

We call definitions of this kind *profiles*.[3] A profile introduces a finite number of new constants that satisfy a given property. Like ordinary definitions, profiles produce conservative extensions, but unlike ordinary definitions, the constants introduced by a profile cannot generally be eliminated. A profile can be viewed as a generalization of a definition since any definition can be expressed as a profile.

Profiles are very useful for introducing new machinery into a theory. For example, a profile can be used to introduce a collection of objects plus a set of operations on the objects—what is called an "algebra" in mathematics and an "abstract datatype" in computer science. The new machinery will not compromise the original machinery of T because the resulting extension T' of T will be conservative. Since T' is a conservative extension of T, any reasoning performed in T could just as well have been performed in T'. Thus the availability of T' normally makes T obsolete.

Making profiles in the naive infrastructure leads to theory objects which store obsolete theories. The way of implementing definitions by extending theories in place would work just as well for profiles. As with definitions, extending theories in place could cause some interpretations to break. A combination of the second and third basic solutions to the problem given above for definitions could be used for profiles. The first basic solution is not applicable because profiles do not generally have the eliminability property of definitions.

C. Theory Extensions Suppose that $\mathbf{S}$ contains two theory objects $\mathbf{T}$ and $\mathbf{T}'$ with $\mathsf{thy}(\mathbf{T}) \leq \mathsf{thy}(\mathbf{T}')$. In most cases (but not all), one would want every theorem object installed in $\mathbf{T}$ to also be installed in $\mathbf{T}'$. The naive infrastructure does not have this capability. That is, there is no support for having theorem objects installed in a theory object to automatically be installed in preselected extensions of the theory object. An intertheory infrastructure should guarantee that, for each theory object $\mathbf{T}$ and each preselected extension $\mathbf{T}'$ of $\mathbf{T}$, every theorem, definition, and profile installed in $\mathbf{T}$ is also installed in $\mathbf{T}'$.

D. Theory Copies The naive infrastructure does not allow the infrastructure state to contain two theory objects storing the same theory. As a consequence, it is not possible to add a copy of a theory object to the infrastructure state. We will see in section 5 that creating copies of a theory object is a useful modularization technique.

[3] Profiles are called *constant specifications* in [13] and *constraints* in [15].

3.3 Requirements

Our analysis of the naive intertheory infrastructure suggests that the intertheory infrastructure should satisfy the following requirements:

R1 *The infrastructure enables theories and interpretations to be stored.*

R2 *Known theorems of a theory can be stored with the theory.*

R3 *Definitions can be made in a theory by extending the theory in place.*

R4 *Profiles can be made in a theory by extending the theory in place.*

R5 *Theorems, definitions, and profiles installed in a theory are automatically installed in certain preselected extensions of the theory.*

R6 *An interpretation of T_1 in T_2 can be extended in place to an interpretation of T_1' in T_2' if T_i is extended to T_i' by definitions or profiles for $i = 1, 2$.*

R7 *A copy of a stored theory can be created and then developed independently from the original theory.*

The naive infrastructure satisfies only requirements **R1** and **R2**. The IMPS intertheory infrastructure satisfies all of the requirements except **R4** and **R7**.

4 The Intertheory Infrastructure

This section presents an intertheory infrastructure that satisfies all seven requirements in section 3.3. It is the same as the naive infrastructure except that the infrastructure objects and operations are different. That is, the infrastructure state is a set of infrastructure objects, is initially the empty set, and is changed by the application of infrastructure operations which add new objects to the state or modify objects already in the state. As in the naive infrastructure, let **S** denote the infrastructure state.

4.1 Objects

There are five kinds of infrastructure objects. The first four are defined simultaneously as follows:

1. A *theory object* is a tuple $\mathbf{T} = (n, L_0, \Gamma_0, L, \Gamma, \Delta, \sigma, \mathcal{N})$ where:
   (a) n is a string called the *name* of $\mathbf{T}$. It is denoted by $[\mathbf{T}]$.
   (b) L_0 and L are languages such that $L_0 \subseteq L$. L_0 and L are called the *base language* and the *current language* of $\mathbf{T}$, respectively.
   (c) $\Gamma_0 \subseteq \mathcal{S}_{L_0}$ and $\Gamma \subseteq \mathcal{S}_L$ with $\Gamma_0 \subseteq \Gamma$. The members of Γ_0 and Γ are called the *base axioms* and the *current axioms* of $\mathbf{T}$, respectively.
   (d) $\Gamma \subseteq \Delta \subseteq \{A_* \in \mathcal{S}_L : \Gamma \models A_*\}$. The members of Δ are called the *known theorems* of $\mathbf{T}$, and Δ is denoted by $\mathsf{thms}(\mathbf{T})$.

(e) σ is a finite sequence of theorem, definition, and profile objects called the *event history* of $\mathbf{T}$.
(f) $\mathcal{N}$ is a set of names of theory objects called the *principal subtheories* of $\mathbf{T}$. For each $[\mathbf{T}'] \in \mathcal{N}$ with $\mathbf{T}' = (n', L_0', \Gamma_0', L', \Gamma', \Delta', \sigma', \mathcal{N}')$, $L_0' \subseteq L_0$, $\Gamma_0' \subseteq \Gamma_0$, $L' \subseteq L$, $\Gamma' \subseteq \Gamma$, $\Delta' \subseteq \Delta$, and σ' is a subsequence of σ.

The *base theory* of $\mathbf{T}$ is the theory (L_0, Γ_0) and the *current theory* of $\mathbf{T}$, written $\mathsf{thy}(\mathbf{T})$, is the theory (L, Γ).

2. A *theorem object* is a tuple $\mathbf{A} = ([\mathbf{T}], A_*, J)$ where:
   (a) $\mathbf{T}$ is a theory object with $\mathsf{thy}(\mathbf{T}) = (L, \Gamma)$.
   (b) $A_* \in \mathcal{S}_L$. A_* is called the *theorem* of $\mathbf{A}$.
   (c) J is a justification that $\Gamma \models A_*$.
3. A *definition object* is a tuple $\mathbf{D} = ([\mathbf{T}], c_\alpha, E_\alpha, J)$ where:
   (a) $\mathbf{T}$ is a theory object with $\mathsf{thy}(\mathbf{T}) = (L, \Gamma)$.
   (b) c_α is a constant not in L.
   (c) $E_\alpha \in \mathcal{E}_L$. $c_\alpha = E_\alpha$ is called the *defining axiom* of $\mathbf{D}$.
   (d) J is a justification that $\Gamma \models O_*$ where O_* is $(\exists\, x_\alpha \,.\, x_\alpha = E_\alpha)$ and x_α does not occur in E_α.[4] O_* is called the *obligation* of $\mathbf{D}$.
4. A *profile object* is a tuple $\mathbf{P} = ([\mathbf{T}], \mathcal{C}, E_\beta, J)$ where:
   (a) $\mathbf{T}$ is a theory object with $\mathsf{thy}(\mathbf{T}) = (L, \Gamma)$.
   (b) $\mathcal{C} = \{c^1_{\alpha_1}, \ldots, c^m_{\alpha_m}\}$ is a set of constants not in L.
   (c) $E_\beta = (\lambda\, x^1_{\alpha_1} \cdots \lambda\, x^m_{\alpha_m} \,.\, B_*)$ where $x^1_{\alpha_1}, \ldots, x^m_{\alpha_m}$ are distinct variables. $E_\beta(c^1_{\alpha_1}) \cdots (c^m_{\alpha_m})$ is called the *profiling axiom* of $\mathbf{P}$.
   (d) J is a justification that $\Gamma \models O_*$ where O_* is $(\exists\, x^1_{\alpha_1} \cdots \exists\, x^m_{\alpha_m} \,.\, B_*)$. O_* is called the *obligation* of $\mathbf{P}$.

An *event object* is a theorem, definition, or profile object.

Let $\mathbf{T} \leq \mathbf{T}'$ mean $\mathsf{thy}(\mathbf{T}) \leq \mathsf{thy}(\mathbf{T}')$ and $\mathbf{T} \trianglelefteq \mathbf{T}'$ mean $\mathsf{thy}(\mathbf{T}) \trianglelefteq \mathsf{thy}(\mathbf{T}')$. $\mathbf{T}$ is a *structural subtheory* of $\mathbf{T}'$ if one of the following is true:

1. $\mathbf{T} = \mathbf{T}'$.
2. $\mathbf{T}$ is a structural subtheory of a principal subtheory of $\mathbf{T}'$.

$\mathbf{T}$ is a *structural supertheory* of $\mathbf{T}'$ if $\mathbf{T}'$ is a structural subtheory of $\mathbf{T}$.

For a theory object $\mathbf{T} = (n, L_0, \Gamma_0, L, \Gamma, \Delta, \sigma, \mathcal{N})$ and an event object $\mathbf{e}$ whose justification is correct, $\mathbf{T}[\mathbf{e}]$ is the theory object defined as follows:

1. Let $\mathbf{e}$ be a theorem object $([\mathbf{T}'], A_*, J)$. If $\mathbf{T}' \leq \mathbf{T}$, then
$$\mathbf{T}[\mathbf{e}] = (n, L_0, \Gamma_0, L, \Gamma, \Delta \cup \{A_*\}, \sigma \mathbin{\hat{}} \langle \mathbf{e} \rangle, \mathcal{N});$$
otherwise, $\mathbf{T}[\mathbf{e}]$ is undefined.
2. Let $\mathbf{e}$ be a definition object $([\mathbf{T}'], c_\alpha, E_\alpha, J)$. If $\mathbf{T}' \leq \mathbf{T}$ and $c_\alpha \notin L$, then
$$\mathbf{T}[\mathbf{e}] = (n, L_0, \Gamma_0, L \cup \{c_\alpha\}, \Gamma \cup \{A_*\}, \Delta \cup \{A_*\}, \sigma \mathbin{\hat{}} \langle \mathbf{e} \rangle, \mathcal{N})$$
where A_* is the defining axiom of $\mathbf{e}$; otherwise, $\mathbf{T}[\mathbf{e}]$ is undefined.

[4] In $\mathbf{C}$, $\Gamma \models (\exists\, x_\alpha \,.\, x_\alpha = E_\alpha)$ always holds and so no justification is needed, but in other logics such as LUTINS a justification is needed since $\Gamma \models (\exists\, x_\alpha \,.\, x_\alpha = E_\alpha)$ will not hold if E_α is undefined.

3. Let $\mathbf{e}$ be a profile object $([\mathbf{T}'], \mathcal{C}, E_\beta, J)$. If $\mathbf{T}' \leq \mathbf{T}$ and $\mathcal{C} \cap L = \emptyset$, then

$$\mathbf{T}[\mathbf{e}] = (n, L_0, \Gamma_0, L \cup \mathcal{C}, \Gamma \cup \{A_*\}, \Delta \cup \{A_*\}, \sigma \hat{} \langle \mathbf{e} \rangle, \mathcal{N})$$

where A_* is the profiling axiom of $\mathbf{e}$; otherwise, $\mathbf{T}[\mathbf{e}]$ is undefined.

An event history σ is *correct* if the justification in each member of σ is correct. For a correct event history σ, $\mathbf{T}[\sigma]$ is defined by:

1. Let $\sigma = \langle\rangle$. Then $\mathbf{T}[\sigma] = \mathbf{T}$.
2. Let $\sigma = \sigma' \hat{} \langle \mathbf{e} \rangle$. If $(\mathbf{T}[\sigma'])[\mathbf{e}]$ is defined, then $\mathbf{T}[\sigma] = (\mathbf{T}[\sigma'])[\mathbf{e}]$; otherwise, $\mathbf{T}[\sigma]$ is undefined.

Let the *base* of $\mathbf{T}$, written $\mathsf{base}(\mathbf{T})$, be the theory object

$$(n\_\mathsf{base}, L_0, \Gamma_0, L_0, \Gamma_0, \Gamma_0, \langle\rangle, \emptyset).$$

$\mathbf{T}$ is *proper* if the following conditions are satisfied:

1. Its event history σ is correct.
2. $\mathsf{thy}(\mathbf{T}) = \mathsf{thy}(\mathsf{base}(\mathbf{T})[\sigma])$.
3. $\mathsf{thms}(\mathbf{T}) = \mathsf{thms}(\mathsf{base}(\mathbf{T})[\sigma])$.

Lemma 2. *If $\boldsymbol{T}$ is a proper theory object, then A_* is a known theorem of $\boldsymbol{T}$ iff A_* is a base axiom of $\boldsymbol{T}$ or a theorem, defining axiom, or profiling axiom of an event object in the event history of $\boldsymbol{T}$.*

Proof. Follows immediately from the definitions above.

Theorem 3. *If $\boldsymbol{T}$ is a proper theory object, then* $\mathsf{base}(\boldsymbol{T}) \trianglelefteq \boldsymbol{T}$.

Proof. Since $\mathbf{T}$ is proper, the event history σ of $\mathbf{T}$ is correct and $\mathsf{thy}(\mathbf{T}) = \mathsf{thy}(\mathsf{base}(\mathbf{T})[\sigma])$. We will show $\mathsf{base}(\mathbf{T}) \trianglelefteq \mathbf{T}$ by induction on $|\sigma|$, the length of σ.

Basis. Assume $|\sigma| = 0$. Then $\mathsf{thy}(\mathbf{T}) = \mathsf{thy}(\mathsf{base}(\mathbf{T}))$ and so $\mathsf{base}(\mathbf{T}) \trianglelefteq \mathbf{T}$ is obviously true.

Induction step. Assume $|\sigma| > 0$. Suppose $\sigma = \sigma' \hat{} \langle \mathbf{e} \rangle$. By the induction hypothesis, $\mathsf{base}(\mathbf{T}) \trianglelefteq \mathsf{base}(\mathbf{T})[\sigma']$. We claim $\mathsf{base}(\mathbf{T})[\sigma'] \trianglelefteq (\mathsf{base}(\mathbf{T})[\sigma'])[\mathbf{e}]$. If $\mathbf{e}$ is a theorem object, then clearly $\mathsf{thy}(\mathsf{base}(\mathbf{T})[\sigma']) = \mathsf{thy}((\mathsf{base}(\mathbf{T})[\sigma'])[\mathbf{e}])$ and so $\mathsf{base}(\mathbf{T})[\sigma'] \trianglelefteq (\mathsf{base}(\mathbf{T})[\sigma'])[\mathbf{e}]$. If $\mathbf{e}$ is a definition or profile object, then $\mathsf{base}(\mathbf{T})[\sigma'] \trianglelefteq (\mathsf{base}(\mathbf{T})[\sigma'])[\mathbf{e}]$ by the justification of $\mathbf{e}$. Therefore, $\mathsf{base}(\mathbf{T}) \trianglelefteq \mathbf{T}$ follows by part (2) of Lemma 1. □

We will now define the fifth and last infrastructure object: An *interpretation object* is a tuple $\mathbf{I} = ([\mathbf{T}], [\mathbf{T}'], \Phi, J)$ where:

1. $\mathbf{T}$ is a theory object called the *source theory* of $\mathbf{I}$.
2. $\mathbf{T}'$ is a theory object called the *target theory* of $\mathbf{I}$.
3. Φ is a translation.
4. J is a justification that Φ is an interpretation of $\mathsf{thy}(\mathsf{base}(\mathbf{T})[\sigma])$ in $\mathsf{thy}(\mathsf{base}(\mathbf{T}')[\sigma'])$ where σ and σ' are initial segments of the event histories of $\mathbf{T}$ and $\mathbf{T}'$, respectively.

4.2 Operations

The infrastructure design includes ten operations.

There are operations for creating the infrastructure objects:

1. Given a string n, a language L, a set Γ of sentences, and theory objects $\mathbf{T}^i = (n^i, L^i_0, \Gamma^i_0, L^i, \Gamma^i, \Delta^i, \sigma^i, \mathcal{N}^i) \in \mathbf{S}$ for $i = 1, \ldots, m$ as input, let
   (a) $L'_0 = L^1_0 \cup \cdots \cup L^m_0$.
   (b) $\Gamma'_0 = \Gamma^1_0 \cup \cdots \cup \Gamma^m_0$.
   (c) $L' = L^1 \cup \cdots \cup L^m$.
   (d) $\Gamma' = \Gamma^1 \cup \cdots \cup \Gamma^m$.
   (e) $\Delta' = \Delta^1 \cup \cdots \cup \Delta^m$.
   (f) $\sigma' = \sigma^1 \hat{} \cdots \hat{} \sigma^m$.

   If

   $$\mathbf{T} = (n, L \cup L'_0, \Gamma \cup \Gamma'_0, L \cup L', \Gamma \cup \Gamma', \Gamma \cup \Delta', \sigma', \{[\mathbf{T}_1], \ldots, [\mathbf{T}_m]\})$$

   is a theory object and $n \neq [\mathbf{T}']$ for any theory object $\mathbf{T}' \in \mathbf{S}$, then create-thy-obj adds $\mathbf{T}$ to $\mathbf{S}$; otherwise, the operation fails.
2. Given a theory object $\mathbf{T} \in \mathbf{S}$, a sentence A_*, and a justification J as input, if $\mathbf{A} = ([\mathbf{T}], A_*, J)$ is a theorem object, then create-thm-obj adds $\mathbf{A}$ to $\mathbf{S}$; otherwise, the operation fails.
3. Given a theory object $\mathbf{T} \in \mathbf{S}$, a constant c_α, an expression E_α, and a justification J as input, if $\mathbf{D} = ([\mathbf{T}], c_\alpha, E_\alpha, J)$ is a definition object, then create-def-obj adds $\mathbf{D}$ to $\mathbf{S}$; otherwise, the operation fails.
4. Given a theory object $\mathbf{T} \in \mathbf{S}$, a set $\mathcal{C}$ of constants, an expression E_β, and a justification J as input, if $\mathbf{P} = ([\mathbf{T}], \mathcal{C}, E_\beta, J)$ is a profile object, then create-pro-obj adds $\mathbf{P}$ to $\mathbf{S}$; otherwise, the operation fails.
5. Given two theory objects $\mathbf{T}, \mathbf{T}' \in \mathbf{S}$, a translation Φ, and a justification J as input, if $\mathbf{I} = ([\mathbf{T}], [\mathbf{T}'], \Phi, J)$ is an interpretation object, then create-int-obj adds $\mathbf{I}$ to $\mathbf{S}$; otherwise, the operation fails.

There are operations for installing theorem, definition, and profile objects in theory objects:

1. Given a theorem object $\mathbf{A} = ([\mathbf{T}_0], A_*, J) \in \mathbf{S}$ and a theory object $\mathbf{T}_1 \in \mathbf{S}$, if $\mathbf{T}_0 \leq \mathbf{T}_1$, then install-thm-obj replaces every structural supertheory $\mathbf{T}$ of $\mathbf{T}_1$ in $\mathbf{S}$ with $\mathbf{T}[\mathbf{A}]$; otherwise, the operation fails.
2. Given a definition object $\mathbf{D} = ([\mathbf{T}_0], c_\alpha, E_\alpha, J) \in \mathbf{S}$ and a theory object $\mathbf{T}_1 \in \mathbf{S}$, if $\mathbf{T}_0 \leq \mathbf{T}_1$ and $\mathbf{T}[\mathbf{D}]$ is defined for every structural supertheory $\mathbf{T}$ of $\mathbf{T}_1$ in $\mathbf{S}$, then install-def-obj replaces every structural supertheory $\mathbf{T}$ of $\mathbf{T}_1$ in $\mathbf{S}$ with $\mathbf{T}[\mathbf{D}]$; otherwise, the operation fails.
3. Given a profile object $\mathbf{P} = ([\mathbf{T}_0], \mathcal{C}, E_\beta, J) \in \mathbf{S}$ and a theory object $\mathbf{T}_1 \in \mathbf{S}$, if $\mathbf{T}_0 \leq \mathbf{T}_1$ and $\mathbf{T}[\mathbf{P}]$ is defined for every structural supertheory $\mathbf{T}$ of $\mathbf{T}_1$ in $\mathbf{S}$, then install-pro-obj replaces every structural supertheory $\mathbf{T}$ of $\mathbf{T}_1$ in $\mathbf{S}$ with $\mathbf{T}[\mathbf{P}]$; otherwise, the operation fails.

There are operations to extend an interpretation object and to copy a theory object:

1. Given an interpretation object $\mathbf{I} = ([\mathbf{T}], [\mathbf{T}'], \Phi, J) \in \mathbf{S}$, a translation Φ', and a justification J' as input, if Φ' extends Φ and $\mathbf{I}' = ([\mathbf{T}], [\mathbf{T}'], \Phi', J')$ is an interpretation object, then extend-int replaces $\mathbf{I}$ in $\mathbf{S}$ with $\mathbf{I}'$; otherwise, the operation fails.
2. Given a string n and a theory object

$$\mathbf{T} = (n', L_0, \Gamma_0, L, \Gamma, \Delta, \sigma, \mathcal{N}) \in \mathbf{S}$$

as input, if $n \neq [\mathbf{T}']$ for any theory object $\mathbf{T}' \in \mathbf{S}$, then create-thy-copy adds the theory object

$$\mathbf{T}' = (n, L_0, \Gamma_0, L, \Gamma, \Delta, \sigma, \mathcal{N})$$

to $\mathbf{S}$; otherwise, the operation fails.

The infrastructure operations guarantee that the following theorem holds:

Theorem 4. *If the justification of every event object in $\boldsymbol{S}$ is correct, then:*

1. *Every object in $\boldsymbol{S}$ is a well-defined theory, theorem, definition, profile, or interpretation object.*
2. *Every theory object in $\boldsymbol{S}$ is proper.*
3. *Distinct theory objects in $\boldsymbol{S}$ have distinct names.*

Some remarks about the intertheory infrastructure:

1. Theory and interpretation objects are modifiable, but event objects are not.
2. The event history of a theory object records how the theory object is constructed from its base theory.
3. The theory stored in a theory object $\mathbf{T}$ extends all the theories stored in the principal subtheories of $\mathbf{T}$.
4. Theorem, definition, and profile objects installed in a theory $\mathbf{T}$ in $\mathbf{S}$ are automatically installed in every structural supertheory of $\mathbf{T}$ in $\mathbf{S}$.
5. The infrastructure allows definitions and profiles to be made in a theory object $\mathbf{T}$ both by modifying $\mathbf{T}$ using install-def-obj and install-prof-obj and by creating an extension of $\mathbf{T}$ using create-thy-obj.
6. By Theorem 2, if Φ is a translation from $\mathsf{thy}(\mathbf{T})$ to $\mathsf{thy}(\mathbf{T}')$ which maps the base axioms of $\mathbf{T}$ to known theorems of $\mathbf{T}'$, then Φ is an interpretation of $\mathsf{thy}(\mathbf{T})$ in $\mathsf{thy}(\mathbf{T}')$.
7. The interpretation stored in an interpretation object is allowed to be incomplete. It can be extended as needed using extend-int.

5 Some Applications

5.1 Theory Development System

The intertheory infrastructure provides a strong foundation on which to build a system for developing axiomatic theories. The infrastructure operations enable theories and interpretations to be created and extended. Many additional operations can be built on top of the ten infrastructure operations. Examples include operations for transporting theorems, definitions, and profiles from one theory to another and for instantiating theories.

Given a theorem object $\mathbf{A} = ([\mathbf{T}_0], A_*, J_0)$ installed in $\mathbf{T} \in \mathbf{S}$ and an interpretation object $\mathbf{I} = ([\mathbf{T}], [\mathbf{T}'], \Phi, J) \in \mathbf{S}$ as input, the operation transport-thm-obj would invoke create-thm-obj and install-thm-obj to create a new theorem object $([\mathbf{T}'], \Phi(A_*), J')$ and install it in $\mathbf{T}'$. The justification J' would be formed from J_0 and J.

Given a constant d_β, a definition object $\mathbf{D} = ([\mathbf{T}_0], c_\alpha, E_\alpha, J)$ installed in $\mathbf{T} \in \mathbf{S}$, and an interpretation object $\mathbf{I} = ([\mathbf{T}], [\mathbf{T}'], \Phi, J) \in \mathbf{S}$ as input, if $\Phi(\alpha) = \beta$ and d_β is not in the current language of $\mathbf{T}'$, the operation transport-def-obj would invoke create-def-obj and install-def-obj to create a new definition object $([\mathbf{T}'], d_\beta, \Phi(E_\alpha), J')$ and install it in $\mathbf{T}'$; otherwise, the operation fails. The justification J' would be formed from J_0 and J. An operation transport-pro-obj could be defined similarly.

Given theory objects $\mathbf{T}, \mathbf{T}' \in \mathbf{S}$ and an interpretation object $\mathbf{I} = ([\mathbf{T}_0], [\mathbf{T}'_0], \Phi, J) \in \mathbf{S}$ as input, if $\mathbf{T}_0 \leq \mathbf{T}$ and $\mathbf{T}'_0 \leq \mathbf{T}'$, the operation instantiate-thy would invoke create-thy-obj to create a new theory object $\mathbf{T}''$ and create-int-obj to create a new interpretation object $\mathbf{I}' = ([\mathbf{T}], [\mathbf{T}''], \Phi', J)$ such that:

- $\mathbf{T}''$ is an extension of $\mathbf{T}'$ obtained by "instantiating" $\mathbf{T}_0$ in $\mathbf{T}$ with $\mathbf{T}'$. How $\mathbf{T}'$ is cemented to the part of $\mathbf{T}$ outside of $\mathbf{T}_0$ is determined by Φ. The constants of $\mathbf{T}$ which are not in $\mathbf{T}_0$ may need to be renamed and retagged to avoid conflicts with the constants in $\mathbf{T}'$.
- Φ' is an interpretation of $\mathsf{thy}(\mathbf{T})$ in $\mathsf{thy}(\mathbf{T}'')$ which extends Φ.

For further details, see [7].

This notion of theory instantiation is closely related to the notion of theory instantiation proposed by Burstall and Goguen [2]; in both approaches a theory is instantiated via an interpretation. However, in our approach, any theory can be instantiated with respect to any of its subtheories. In the Burstall-Goguen approach, only "parameterized theories" can be instantiated and only with respect to the explicit parameter of the parameterized theory.

5.2 Foundational Theory Development System

A theory development system is *foundational* if every theory developed in the system is consistent relative to one or more "foundational" theories which are known or regarded to be consistent. Since the operations for installing theorems, definitions, and profiles in a theory always produce conservative extensions of the original theory by Theorem 3, these operations preserve consistency. Therefore, a foundational theory development system can be implemented on top of the infrastructure design by simply using a new operation for creating theory objects that is successful only when the theory stored in the object is consistent relative to one of the foundational theories.

The new operation can be defined as follows. Suppose $\mathbf{T}^*$ is a foundational theory. Given a string n, a language L, a set Γ of sentences, theory objects $\mathbf{T}_1, \ldots, \mathbf{T}_m \in \mathbf{S}$, a translation Φ, and a justification J as input, if J is a justification that Φ is an interpretation of $T = (L, \Gamma)$ in $\mathsf{thy}(\mathbf{T}^*)$, the new operation would invoke create-thy-obj on $(n, L, \Gamma, \{[\mathbf{T}_1], \ldots, [\mathbf{T}_m]\})$ to create a theory object $\mathbf{T}$ and then invoke create-int-obj on $([\mathbf{T}], [\mathbf{T}^*], \Phi, J)$ to create an interpretation object $\mathbf{I}$; otherwise the operation fails. If the operation is successful and J is correct, then $\mathsf{thy}(\mathbf{T})$ would be consistent relative to $\mathsf{thy}(\mathbf{T}^*)$ by Theorem 1.

5.3 Encapsulated Theory Development

Proving a theorem in a theory may require introducing several definitions and proving several lemmas in the theory that would not be useful after the theorem is proved. Such "local" definitions and lemmas would become logical clutter in the theory. One strategy for handling this kind of clutter is to encapsulate local development in a auxiliary theory so that it can be separated from the development of the main theory. The infrastructure design makes this encapsulation possible.

Suppose that one would like to prove a theorem in a theory stored in theory object $\mathbf{T}$ using some local definitions and lemmas. One could use create-thy-copy to create a copy $\mathbf{T}'$ of $\mathbf{T}$ and create-int-obj to create a interpretation object $\mathbf{I}$ storing the identity interpretation of $\mathsf{thy}(\mathbf{T}')$ in $\mathsf{thy}(\mathbf{T})$. Next the needed local definitions and lemmas could be installed as definition and theorem objects in $\mathbf{T}'$. Then the theorem could be proved and installed as a theorem object in $\mathbf{T}'$. Finally, the theorem could be transported back to $\mathbf{T}$ using the interpretation stored in $\mathbf{I}$. The whole local development needed to prove the theorem would reside in $\mathbf{T}'$ completely outside of the development of $\mathbf{T}$.

A different way to encapsulate local theory development is used in the ACL2 theorem prover [15].

5.4 Sequent-Style Proof System

A goal-oriented sequent-style proof system can be built on top of the intertheory infrastructure. A *sequent* would have the form $\mathbf{T} \rightarrow A_*$ where $\mathbf{T}$ is a theory object called the *context* and A_* is a sentence in the current language of $\mathbf{T}$

called the *assertion*. The system would include the usual inference rules of a sequent-style proof system plus rules to:

- Install a theorem, definition, or profile into the context of a sequent.
- Transport a theorem, definition, or profile from a theory object to the context of a sequent.

Some of the proof rules, such as the deduction rule, would add or remove axioms from the context of a sequent, thereby defining new theory objects. The proof rules for the rules of universal generalization (∀-introduction) and existential instantiation (∃-elimination) would be implemented by installing a profile in the context of a sequent.

A sentence A_* in the current language of a theory object $\mathbf{T}$ would be proved as follows. create-thy-copy would be used to create a copy $\mathbf{T}'$ of $\mathbf{T}$ and create-int-obj would be used to create a interpretation object $\mathbf{I}$ storing the identity interpretation of $\mathsf{thy}(\mathbf{T}')$ in $\mathsf{thy}(\mathbf{T})$. Then the sequent $\mathbf{T}' \rightarrow A_*$ would be proved, possibly with the help of local or imported definitions and lemmas. The contexts created in the course of the proof would be distinct supertheories of $\mathbf{T}'$. A theorem or definition installed in a context appearing in some part of the proof would be available wherever else the context appeared in the proof.

When the proof is finished, A_* would be installed as a theorem object in $\mathbf{T}'$. The theorem could be then transported back to $\mathbf{T}$ using the interpretation stored in $\mathbf{I}$. The theory objects needed for the proof—$\mathbf{T}'$ and its supertheories—would be separated from $\mathbf{T}$ and the other theory objects in $\mathbf{S}$.

Acknowledgments

Many of the ideas in this paper originated in the design and implementation of IMPS done jointly by Dr. Joshua Guttman, Dr. Javier Thayer, and the author.

References

1. P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Academic Press, 1986.
2. R. Burstall and J. Goguen. The semantics of Clear, a specification language. In *Advanced Course on Abstract Software Specifications*, volume 86 of *Lecture Notes in Computer Science*, pages 292–332. Springer-Verlag, 1980.
3. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
4. H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.
5. W. M. Farmer. A partial functions version of Church's simple theory of types. *Journal of Symbolic Logic*, 55:1269–91, 1990.
6. W. M. Farmer. A simple type theory with partial functions and subtypes. *Annals of Pure and Applied Logic*, 64:211–240, 1993.
7. W. M. Farmer. A general method for safely overwriting theories in mechanized mathematics systems. Technical report, The MITRE Corporation, 1994.

8. 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.
9. 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.
10. W. M. Farmer, J. D. Guttman, and F. J. Thayer. Little theories. In D. Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 567–581. Springer-Verlag, 1992.
11. W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11:213–248, 1993.
12. J. A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-99-9, SRI International, August 1988.
13. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
14. N. Hamilton, R. Nickson, O. Traynor, and M. Utting. Interpretation and instantiation of theories for reasoning about formal specifications. In M. Patel, editor, *Proceedings of the Twentieth Australasian Computer Science Conference*, volume 19 of *Australian Computer Science Communications*, pages 37–45, 1997.
15. M. Kaufmann and J S. Moore. Structured theory development for a mechanized logic. Available at `http://www.cs.utexas.edu/users/moore/publications/acl2-papers.html`, 1999.
16. R. Nakajima and T. Yuasa, editors. *The* IOTA *Programming System*, volume 160 of *Lecture Notes in Computer Science*. Springer-Verlag, 1982.
17. 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.
18. J. Rushby, F. von Henke, and S. Owre. An introduction to formal specification and verification using EHDM. Technical Report SRI-CSL-91-02, SRI International, 1991.
19. J. R. Shoenfield. *Mathematical Logic*. Addison-Wesley, 1967.
20. D. Smith. KIDS: A knowledge-based software development system. In M. Lowry and R. McCartney, editors, *Automating Software Design*, pages 483–514. MIT Press, 1991.
21. Y. Srinivas and R. Jullig. Specware: Formal support for composing software. In *Proceedings of the Conference on Mathematics of Program Construction*, 1995.