2OBJ: A Metalogical Framework Theorem Prover Based on Equational Logic [and Discussion]

Joseph Goguen, Andrew Stevens, Hendrik Hilberdink, Keith Hobley, W. A. Hunt and T. F. Melham

Phil. Trans. R. Soc. Lond. A 1992 **339**, 69-86
doi: 10.1098/rsta.1992.0026

2OBJ: a metalogical framework theorem prover based on equational logic

By Joseph Goguen[1], Andrew Stevens, Hendrik Hilberdink and Keith Hobley

Programming Research Group, Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, U.K., and [1]*SRI International, Menlo Park, California, U.S.A.*

This paper describes 2OBJ, a tactic-based generic theorem prover that encodes object logics into equational logic via an abstract data type of object logic sentences and proofs. 2OBJ is built upon OBJ3, a term rewriting implementation of (order sorted conditional) equational logic. Because object logic proofs are explicitly represented, 2OBJ can not only reason with them, but also about them, as in arguments by symmetry and other metalogical devices of ordinary mathematics; this motivates the 'meta' of 'metalogical' in the title. First-order equational logic has advantages in simplicity and efficiency over more complex framework logics, such as intuitionistic higher-order type theory, and also facilitates the definition of tactic languages. In addition, 2OBJ benefits from OBJ3's powerful parametrized module system, and it has a convenient X window user interface. The paper concludes with a sketch of some semantic foundations based upon ruled parchments, charters, and institutions.

1. Introduction

There is a population explosion among the logical systems used in many areas of computing science, including hardware description and verification. Unfortunately, there does not seem to be any one logical system that is clearly better than all the rest, even for such a specialized area as hardware design. This situation has inspired the development of so-called 'logical frameworks', which are systems within which theorem proving for a range of 'object logics' can be implemented by encoding into a logical system called a *framework logic*, usually a powerful type theory. This paper argues for using a very simple framework logic, namely order sorted equational logic, because of several practical advantages.

1. It is easier to implement an equational logic efficiently, because of its well-explored technology base with many efficient implementation techniques, such as term rewriting and unification.

2. It is easier to encode logics into equational logic. Indeed, the basic idea is very simple: formulae and proofs are terms; and deduction is term rewriting (in general, conditional).

3. When proof systems are encoded using an abstract data type of proofs, it is easier to use and justify 'reasoning by symmetry' and other metalogical devices that can often greatly simplify proofs.

4. It is also easier to reason about properties of the logical framework itself, such as the correctness of a given implementation technique, or of a given encoding.

5. Our framework logic (equational logic) is executable, so that a tactic programming language can be implemented directly in it.

The 2OBJ system dramatically illustrates the first and last points by building on the already developed OBJ3 term rewriting system, which is rigorously based on (conditional order sorted) equational logic, and provides a number of other helpful facilities, such as: (1) an efficient implementation of term rewriting; (2) rewriting modulo associativity and/or commutativity; (3) a powerful parametrized module mechanism, as well as multiple inheritance for modules, and a number of built-in modules for basic abstract data types; (4) a distinction between executable and property-oriented modules, with so-called 'views' for asserting that a given module satisfies a given set of properties; (5) module expressions for describing and building systems out of given modules, using views as bindings for the interfaces of parametrized modules; (6) easy access to programs in LISP (or C), to provide efficient 'built-in' implementations for specific decision procedures (such as propositional logic); (7) user definable prefix, infix, and mixfix (as in `if_then_else_fi`) syntax; and (8) a convenient and expressive notion of subtype, provided by the order sorted framework logic.

2OBJ has an X window based graphical user interface for applying rules of deduction with OBJ3 under the control of a tactic language. The name 2OBJ was chosen because the system uses OBJ to implement both object and meta-level logics.

OBJ3's ability to 'build-in' new features through the underlying LISP is useful for implementing tactic languages. As far as we know, no other logical framework provides tactic languages, although these can save much effort in actually doing proofs. OBJ3's parametrized module facility provides a higher-order capability, which compensates for its lack in the underlying first order equational logic, and which also enhances the reusability of (parts of) encodings. See Futatsugi *et al.* (1985) and Futatsugi *et al.* (1987) for more information on OBJ, and Goguen *et al.* (1992) for the latest information on OBJ3.

The basic approach to encoding an object logic in 2OBJ is to regard its proofs as an abstract data type, which is then manipulated by its rules of deduction; we use the word '*frame*' for such an encoding. The connection between equational logic, abstract data types, and term rewriting is now so well known that it should need no further elaboration as motivation for building a logical framework on a system like OBJ3. This paper is focused on implementation techniques rather than on their logical foundations, but we do briefly describe our approach to foundations, based on representing logical systems by parchments enriched with 'rulings' to capture inference rules.

This work is part of the SSVE project, supported jointly by SERC and DTI; this collaboration between Oxford and RHBNC aims to develop a uniform environment for hardware specification, simulation and verification, that can be used with a variety of hardware description languages, by translating them into a rigorous kernel language, called FUNNEL. In this context, 2OBJ is a 'logical funnel' supporting multiple logical systems for hardware design, such as temporal logic, first-order logic, higher-order logic, propositional logic, and second-order equational logic. However, 2OBJ has many other applications, for example, in software development.

2. Mechanized theorem provers and logical frameworks

Automatic theorem provers provide fully automated proof search, guided primarily by the axioms and lemmas available to the system. The proofs found by such systems depend on the search strategy implemented, although in practice users can often get the behaviour that they want by setting proof search parameters or by providing hints about the likely role of particular lemmas and axioms. Good examples of such systems are the Otter resolution theorem prover (McCune 1989), Boyer & Moore's NQTHM (Boyer & Moore 1980), and Bundy's Clam (Bundy *et al.* 1990). By contrast, mechanized theorem provers like 2OBJ are *proof assistants* that support the construction of proofs by human beings, by allowing commonly used proof procedures to be programmed as proof tactics. A *proof checker* is a proof assistant without a tactic language. (Typical current generation automatic theorem provers are of limited use as proof assistants, and are also difficult to integrate into systems other than those for which they were designed. In principle, it is possible to write tactics that implement the proof search strategies of automatic theorem provers. But in practice, this would be very hard to do in a worthwhile way.)

The conventional architecture for mechanized theorem provers follows the Edinburgh LCF system from the late 1970s (Gordon *et al.* 1979) in implementing a logic as an abstract data type (ADT) of proofs (or theorems) written directly in a programming language (usually LISP or ML), so that the inference rules of the logic are primitive operations of the ADT. This permits tactics to be implemented elegantly as programs that apply these primitives. The soundness of such proofs can be rigorously enforced through the type discipline of the programming language. HOL (Gordon 1985) and Nuprl (Constable *et al.* 1986) are well-known systems with this architecture.

Unfortunately, the LCF architecture has some significant drawbacks in practice. First, it implements just one hard-wired logical system, and cannot easily be configured to implement other logics. Second, mechanizing a logic in this way restricts mechanized reasoning to the object level, so that it is not possible to reason about (rather than within) the implemented logic.

Motivated by these problems, recent work in mechanized theorem proving (Constable & Basin 1991; Harper *et al.* 1987; Matthews *et al.* 1991) has focused on *logical frameworks*, in which logics are mechanized indirectly by encoding them within a suitable *framework logic* that is mechanized in the conventional way. Proofs in the encoding logic, which is called the *object logic*, are constructed in encoded form by reasoning in the framework logic. Such systems mechanize a logic by introducing axioms to define the logic's syntax and inference rules. Given a suitable framework logic, it is usually relatively easy to write the axioms for a frame from a pencil and paper presentation of an object logic, and to verify these axioms against it, because the axioms are usually close to those in the pencil and paper presentation.

Thus logical frameworks have the potential to mechanize logical systems 'on demand' for experimentation or for the specialized needs of particular users. This flexibility can be valuable for hardware design. Furthermore, a logical framework may also permit mechanized support for metareasoning; for example, reasoning by symmetry can be formalized as a theorem about syntactic properties of (encoded) sentences in forms like the following:

$$\vdash (\forall s, c\colon \text{Formula})\,(\forall x, v, v'\colon \mathit{Var})\,(s = \mathit{swap}(v', v, s) \wedge \{v, v'\} \text{ free-in } c)$$
$$\Rightarrow (\text{''}c[v/x] \vdash s\text{''} \Rightarrow \text{''}c[v'/x] \vdash s\text{''}).$$

When the framework logic has a sublogic with an interpreter, then metatheorems can be *executed* to shortcut proof construction. The possibility of metalevel reasoning distinguishes logical frameworks from reconfigurable mechanized theorem provers like EUDOPHILOS (Sawamura *et al.* 1990), which only provide convenient notations for defining logics. Most logical frameworks lack a tactic language, and thus are only proof checkers. Also, they tend to be rather inefficient, because of multiple levels of interpretation. 2OBJ takes a *metalogical framework* approach to encoding logics, in the sense of Constable & Basin (1991), so that object logic syntax and inference rules are encoded into order sorted equational logic, our framework metalogic. Object logic structure is *externalized*, in that the inference rules of the object logic are encoded as primitive operations on an ADT of object logic proofs.

3. A quick review of OBJ3

OBJ has three kinds of entity at its top level: objects, theories, and views. An *object* encapsulates executable code, while a *theory* defines properties that may (or may not) be satisfied by another object or theory. Both objects and theories are *modules*, and consist of declarations and sentences in *order sorted equational logic*, which provides a notion of *subsort* that rigorously supports (a kind of) multiple inheritance, exception handling and overloading. A *view* is a binding of the entities declared in some theory to entities in some other module, and also asserts that the other module satisfies the properties declared in the theory. Theories and views are found in no other implemented language with which we are familiar; however, Standard ML, and perhaps even Ada, have been influenced by this approach.

Modules can import other previously defined modules, and so that an OBJ program is conceptually a *graph* of modules. Modules have *signatures* that introduce new sorts and new operations among both new and imported sorts. *Terms* are built up from operations, respecting their sort declarations. A *reduction* evaluates a given term with respect to a given object, and OBJ supports reduction modulo associativity, commutativity, and/or identity.

OBJ3 provides *parametrized programming*, a technique which helps support design, verification, reuse and maintenance. Modules can be parametrized, and parametrized modules use theories to define both the syntax and semantics of their interfaces. Views indicate how to instantiate a parametrized module with an actual parameter. Actual parameters are modules. *Module expressions* allow complex combinations of already defined modules, including sums, instantiations and transformations; moreover, evaluating a module expression actually constructs the described software (sub)system from the given components. *Default views* can greatly reduce the effort of instantiating modules. Goguen (1990) argues that first-order parametrized programming includes the essential power of higher-order programming.

This kind of module composition is more powerful than the purely functional composition of traditional functional programming, because a single module instantiation can compose together many different functions all at once, in complex ways. For example, a parametrized complex arithmetic module CPXA can be instantiated with any of several real arithmetic modules as actual parameter: single precision reals, CPXA[SP-REAL]; double precision reals, CPXA[DP-REAL]; and multiple precision reals, CPXA[MP-REAL].

Each instantiation involves substituting dozens of functions into dozens of other functions. While something similar is possible in higher-order functional programming by encoding modules as records, it is much less natural, particularly if intended to specify the interface semantics of CPXA. Furthermore, parametrized programming allows the logic to remain first order, so that understanding and verification can be simpler. Typical higher-order functional programming techniques can be implemented with OBJ parametrized modules, often with greater flexibility and clarity (see Goguen 1990).

Although OBJ executable code normally consists of equations that are interpreted as rewrite rules, OBJ objects can also encapsulate LISP code, e.g. to provide efficient built in data types, or to augment the system with new capabilities. This is convenient for specifying, rapid prototyping, and debugging complex data types and algorithms, and for building efficient theorem proving environments. It is also crucial to our implementation of 2OBJ. In addition, OBJ provides rewriting modulo associative, commutative and/or identity equations, as well as user-definable evaluation strategies that allow lazy, eager, and mixed evaluation strategies on an operation-by-operation basis; memoization is also available on an operation-by-operation basis. Finally, OBJ provides user-definable mixfix syntax, to support the notational conventions of particular application domains.

4. The 2OBJ architecture

(*a*) *Encoding logics*

An object logic is encoded in 2OBJ as an ADT with primitive operations for each of its inference rules and logical connectives. OBJ3 parametrized ADTS can be instantiated to encode almost any formal system without significant programming effort. The (meta)user only needs to provide one module defining a term algebra for the concrete syntax, and another for its inference rules as (conditional) equations. The binding behaviour of connectives is defined by equations for the operation bound, for computing which variables are bound in an (encoded) term. These two modules essentially transcribe a conventional pencil and paper presentation of the logic:

1. The abstract syntax of almost any object logic is easily encoded by representing each connective as an operation in a term algebra, and can be given a natural concrete syntax using OBJ3's mixfix syntax.

2. 2OBJ gets the syntactic operations that it needs to define inference rules, substitution, β-reduction, etc., through a *meta-programming interface* to OBJ3. These operations can implement the binding behaviour of encoded terms, so that inference rules can be directly translated into (conditional) OBJ3 equations.

The phrase 'meta-programming interface' above refers to programming language facilities that allow the terms and associated interpreter of a language to be treated as an ADT in the language. A familiar example is LISP's 'eval' function, for evaluating S-expressions as LISP programs. For 2OBJ, the meta-programming interface is an OBJ3 module that provides construction and manipulation primitives for OBJ3 terms and modules, plus rewriting in named modules. The objects FOPC-

SENTENCE and ENCODED-FOPC below show the ease of encoding the unsorted first order predicate calculus in 2OBJ.

```
obj FOPC-SENTENCE is
  sorts Formula Term Var . subsort Var < Term.
  op |A|_._: Var Formula -> Formula [ prec 28 ] .
  op |E|_._: Var Formula -> Formula [ prec 28 ] .
  op_^_: Formula Formula -> Formula [ prec 20 gather ( e & ) ] .
  op_v_: Formula Formula -> Formula [ prec 22 gather ( e & ) ] .
  op_=>_: Formula Formula -> Formula [ prec 25 gather ( e & ) ] .
  op~_: Formula -> Formula [ prec 18 ] .

  *** Import meta-programming interface, Oterm is the
  *** sort of all sequents, and syntactic components of sequents.
  pr OTERMX .
  subsorts Term Formula < Oterm .

  *** The binding characteristics of the operations are defined
  var V : Var . var X : Formula .
  eq bound( ( |A| V . X ) ) = [V] .
  eq bound( ( |E| V . X ) ) = [V] .
endo
  obj ENCODED-FOPC is
  *** We currently use a refinement sequent calculus presentation
  pr SEQUENTS-LANGUAGE[ FOPC-SENTENCES ] .

  *** identify inference rules
  op andi : -> Rule .
  op oril : -> Rule .
  op orir : -> Rule .
  op Ei   : Term -> Rule .
  op ande : Int -> Rule .
  op hyp  : Int -> Rule .

  *** define behaviour of inference rules
  eq andi     ( H |- X ^ Y )      = ( H |- X, H |- Y ).
  eq oril     ( H |- X v Y )      = ( H |- X ) .
  eq orir     ( H |- X v Y )      = ( H |- Y ) .
  eq impi     ( H |- X => Y )     = ( H ; X |- Y ) .
  ceq Ei( T ) (H |- |E| V . Y )   = ( H |- ( Y o [T] / [V] ) )
    if diff( freevars(T), freevars(H |- ( |E| V . Y ))) == nil.

  ceq ande( N ) ( H |- X ) = ( H ; [ Tsubterms(hyp(N,H)) ] |- X )
    if matches( Z ^ Y, hyp(N, H) ) .
  ceq hyp( N ) ( H |- X ) = □
    if X == hyp(N,H) .
  *** ...
endo
```

A term in the proof ADT is a pair $(\Phi \rightarrow \phi)$ denoting a (partial) proof with conclusion ϕ whose completion requires proving the (possibly empty) list Φ of assumptions. These terms are built using just two primitive constructor operations.

1. *Rule application* constructs proof terms for single inference rule applications. Given R (identifying a rule) and G (encoding a sentence), and assuming that the current frame includes an equation $R\ G = \Phi(G, R)$ defining the application of R to G, then the proof term $\Phi(G, R) \to G$ is constructed. For example, given the equations defining rules in `FOPC-ENCODED`, applying the rule `impi` to the goal $\vdash a \Rightarrow b$ gives a partial proof of $\vdash a \Rightarrow b$ from the assumption $a \vdash b$. If there is no applicable equation (i.e. if R does not apply to G), then an exception is returned.

2. *Proof composition* yields a proof term denoting the combination of two or more sub-proofs. Given a term $(\Phi \to \phi)$ encoding a partial proof with assumptions $\Phi = \phi_1, \dots, \phi_n$, and given a list of terms $(\Phi_i \to \phi_i)$ encoding (partial) proofs of those assumptions, then the term $(\Phi_1, \dots, \Phi_n \to \phi)$ is constructed.

The soundness and completeness of these two proof operations follows from the formalism in §5. Rule application constructs proof terms that denote axioms of the object logic, while proof composition implements an amalgamation of combination and composition that suffices for any logic where inference rules have single sentence consequents.

2OBJ's proof ADT is an algebra of proofs in the encoded logic, with proof composition as its key structuring operation. Conventional systems, such as Nuprl (Constable *et al.* 1986; Constable & Basin 1991) or Isabelle (Paulson 1988), identify each inference rule with a corresponding constructor of the proof ADT. Thus these systems construct proofs by composing inference rule primitives, rather than by composing proofs. Therefore, such systems must implement tactic combinators (and hence many tactics) as higher-order functions. By contrast, 2OBJ supports a transparent purely first-order treatment of tactics. This has several important benefits, as discussed in §3*c* below.

Internally, proof terms are derivation trees which may include the subproofs used, as well as the proof's conclusion and assumptions. When two or more proofs are composed, the system constructs the derivation tree for the resulting proof by joining the derivation trees of the components. This extra information is invisible to the tactic interface, but is provided to the user interface (see §4) to support interactive proof editing, for example, to browse through proof structure or to undo some previous step. The level of detail in derivation trees may be reduced by a 'folding' operation, which collapses a subproof tree to just its conclusion and assumptions; this changes the representation of proofs, but not their meaning.

(*b*) *Logical frameworks and meta-level reasoning*

2OBJ takes a *metalogical framework* approach to encoding logics, in the sense of Constable & Basin (1991), using order sorted equational logic as a metalogic for encoding object logic syntax and operations. This *externalizes* object logic structure by encoding inference rules as primitive operations on a proof ADT. By contrast, systems like the Edinburgh Logical Framework (ELF) (Harper *et al.* 1987), Isabelle (Poulson 1987), and λ-Prolog (Miller 1990), *internalize* parts of the object logic, by identifying some object logic structures with corresponding framework logic structures. For example, variable binding in ELF is represented by using lambda abstraction from the framework logic, so that an existential quantifier would be encoded as

$$exists = [t \mid Type][x\colon (t \to Prop)] \dots.$$

In particular, $\exists x{:}Int.P$ would be encoded as $exists[x{:}Int]P$, where $[v{:}T]\phi$ is understood as $\lambda v{:}T.\phi$. (The notation $[v \mid T]\phi$ is similar, except that the instantiation for v is deduced from context.) Also, ELF identifies the object logic consequence relation with that of the framework logic, and encodes object logic inference rules as axioms for a provability predicate, rather than as operations on proofs. For example, the inference rule for implication introduction is encoded as the axiom

$$(\Pi p: Prop)\,(\Pi q: Prop)\,(true(p) \to true(q)) \to (true(\supset(p,q))),$$

where $\Pi x{:}T$ denotes the universal quantification $\forall x{:}T$. A logic encoded in ELF thus 'borrows' its consequence relation and proof structure from the framework logic.

Internalizing object logic structure this way considerably simplifies object logic encoding, because fiddly details of variable binding, proof construction, equality of terms, etc., can be avoided. However, internalization also severely restricts the meta-level reasoning that can be supported. For example, ELF cannot prove meta-theorems by induction over object logic proof structure; rather more seriously, ELF cannot encode logics with consequence relations that differ markedly from its framework logic. 2OBJ and other systems based on externalized encodings (e.g. recent work with Nuprl (Constable & Basin 1991) and LF_0 (Matthews *et al.* 1991)) do not have these restrictions. On the plus side, the ELF encoding supports automatic construction of introduction and elimination rules for connectives, as well as comparatively easy proofs for faithfulness and adequacy, i.e. that every proof in the logic has an equivalent in its encoding and vice versa.

(c) *Proof automation in 2OBJ*

Mechanized theorem proving systems usually support proof automation through the programming language used to implement the system. Typically, the logic is mechanized as an ADT in the implementation language, with the inference rules as primitive operations. Then programs to construct proofs are written in the implementation language by combining inference rule operations. Thus the implementation language is used as a meta-language.

Because 2OBJ's framework logic is implemented in the powerful programming language OBJ3,† tactics can be programmed in its framework logic rather than its implementation language. 2OBJ's tactic programming interface extends the proof ADT to a tactic ADT by adding a tactic application operation, in effect, an interpreter for tactics, based on the following sequential composition combinators:

⟨*First*⟩ `THEN` ⟨*Second*⟩, where ⟨*First*⟩ and ⟨*Second*⟩ are tactics, applies ⟨*First*⟩ and then applies ⟨*Second*⟩ to any subgoals raised by applying ⟨*First*⟩.

⟨*First*⟩ `THENL` ⟨*RuleList*⟩, where ⟨*RuleList*⟩ is list of tactics, applies ⟨*First*⟩ and then applies the elements of ⟨*RuleList*⟩ to the corresponding subgoals raised by ⟨*First*⟩, i.e. the first element of ⟨*RuleList*⟩ is applied to the first subgoal raised, the second element is applied to the second subgoal raised, etc.

Any recursively enumerable sequence of inference rule applications can be programmed with `THEN` and `THENL` by using (recursive) equations written in OBJ3. For example, we can introduce a tactic that repeats a given inference rule a given number of times with the recursive definition

```
eq REPEAT(N, R) = if N == 1
                  then R
                  else R THEN REPEAT(N - 1, R) fi.
```

† OBJ3 was originally designed as an equational prototyping and specification language. The relatively small extra functionality needed to support theorem proving was added as part of the 2OBJ project.

Often one wants to write a tactic whose action depends on the goal to which it is applied, but the sequential composition combinators are not sufficient for such tactics, because they cannot pass goals to tactics. 2OBJ has additional features for tactics of this type.

1. For handling 'do nothing' cases, the identity tactic `idtac` applied to a goal constructs the empty partial proof, containing just that goal.

2. If the tactic being applied is neither a primitive inference rule nor a sequential composition, then the interpreter applies a tactic expansion operation to the tactic and current goal. After this, the interpreter is called again to apply the resulting tactic to the current goal.

For example, we can define a tactic that exhaustively strips off universal quantifiers as follows:

```
op stripUniv : -> Tactic .
vars G : ProofGoal .
eq stripUniv G = if hasleadingunivquant(G)
                  then RuleToIntroUnivQuant THEN stripUniv
                  else idtac fi .
```

The tactic expansion operation has juxtaposition syntax, corresponding to the OBJ3 syntax declaration _ _, and it allows the behaviour of a tactic to depend on its goal. Therefore 2OBJ does not need to implement tactic combinators as higher-order functions, and thus can stay within its first-order equational framework. This makes tactic programming and meta-programming significantly easier and more flexible than the usual higher-order function approach. For example, it is easy to write meta-interpreters for special purpose tactic languages. The operation `insert` defined below executes a composition of tactics, inserting a second tactic when a particular condition is reached. The higher-order functional approach precludes this kind of meta-programming, because it cannot treat tactics as data structures.

```
op insert : Tactic Tactic -> Tactic .
ceq insert(T1 THEN T2, Inttac) G = T1 THEN insert(T2, IntTac)
  if not somecond(G) .
ceq insert(T, IntTac) G = IntTac THEN insert(T, IntTac)
  if somecond(G) .
ceq insert(T, IntTac) G = T
  if not somecond(G) and not isaTHENterm(T) .
```

Another important benefit of embedding the tactic programming language in the framework logic is that machinery for reasoning about object logics can also be used for reasoning about tactics. For example, when using a theorem prover intensively for some application, one often wants to replace a frequently used (or slow running) tactic by a more efficient special purpose inference procedure. In hardware verification, for instance, one might want efficient normalization procedures for certain common circuits. But the specialist procedure and the tactic should be proved equivalent before making the replacement. A theorem prover with a conventional external tactic programming language will require a special proof environment for the tactic programming language. Because this language is typically complex (e.g. a dialect of ML), this can be an extensive undertaking. Thus, such proofs are either done informally, or else are restricted to a subset of the tactic language. In either case, users need to learn a second proof system. Neither of these difficulties applies in 2OBJ.

In principle, any metalogical framework with an interpreter for its framework logic could support a similar embedded tactic programming interface. But in practice, 2OBJ is the only system we know with a workable implementation for its framework logic. For example, the metalogical framework based on Nuprl proposed in Constable & Basin (1991) provides little more than a bare lambda calculus for writing tactics. OBJ3's efficiency as a programming language also minimizes the potential inefficiency of applying an inference step in the (encoded) object logic as several inference steps in the framework logic: OBJ3 executes such inferences at a speed close to that of an implementation in a programming language like ML. However, in systems such as Nuprl and LF_0, the inferences needed to apply an (encoded) inference rule are quite unwieldy, and require non-trivial tactics written in the system's meta-language.

(*d*) *Defining and manipulating theories*

A further benefit of implementing the framework logic in OBJ3 is that its powerful higher-order module system can be used to structure theories and libraries of lemmas. From a logical perspective, the module implementing the proof ADT and tactic interface of 2OBJ is an (equational) theory of encoded proofs. Thus a theory in the object logic can be given as a module extending the object logic module with new atomic proofs for its axioms, as in the following theory of groups:

```
obj GROUP is sort Grp Grp_Var .
 subsorts Grp < Term .
 subsorts Grp_Var < Var .
 subsorts Grp_Var < Grp .
 op _ o _ : Grp Grp -> Grp .
 op e : Grp .
 op inv : Grp -> Grp .
 ops grpid1 grpid2 grpass : ->AxId .
 vars x y z : Grp_Var .
 eq axom(grpid) = ax( * |- |A| x . x o e = x ) .
 eq axiom(grpinv) = ax( * |- |A| x . x o inv(x) = e ) .
 eq axiom(grpass) =
  ax( * |- |A| x . |A| y . |A| z . x o ( y o z ) = ( x o y ) o z ) .
endo
```

Axioms, which are atomic proofs defined with the built-in operator `ax`, appear on the right-hand sides of equations whose left-hand side applies the `axiom` operator to an identifying constant. This indirect approach can ensure that proofs only use those axioms actually given in their defining module.

This theory of groups also illustrates how 2OBJ encodes theories in equational logic. As shown in §3, 2OBJ frames typically use OBJ3 sorts to represent the syntactic kind of encoded object logic terms. For example, in `FOPC-SENTENCE`, the sort of a term indicates whether it encodes a variable `Var` or a non-variable term `Term`. Thus some extra care is needed to encode object logic sorts with OBJ sorts. The usual approach (as in `GROUP`) introduces an OBJ3 sort `S_K` for each possible combination of a sort S and a syntactic kind `K`. Also, for each subsort declaration `subsort K < K'` and sort S a subsort declaration `subsort S_K < S_K'` is introduced. Similarly, to encode an order sorted object theory, we introduce a subsort declaration `subsort S_K < S'_K` for each axiom $S \subset S'$ and syntactic kind

K. For example, an operator $f{:}A \rightarrow C$ of syntactic kind K and argument kind K' would be encoded by an OBJ3 operator

```
op f : A_K' -> C_K .
```

Then OBJ3's sort-checking will ensure that f can only be applied to arguments encoding terms of sort A (or a subsort) and syntactic kind K' (or a subsort). For a more natural looking presentation, we can name the sorts S_K just S when there is no syntactic kind K' that contains K. Thus, in the GROUP example, to encode Vars or Terms with sort *Grp*, we introduce the OBJ3 sort Grp_Var for encoding variables of sort *Grp*, but use the sort Grp for encoding Terms of sort *Grp*. Alternatively, we can use OBJ3's 'sort-constraint' mechanism to enforce syntactic constraints, and then use OBJ3 sorts exclusively to encode sort information. Though somewhat more complicated, this is useful when there are many syntactic kinds or when the constraints on object syntax cannot be expressed with OBJ3 sorts.

Theories defined this way can be combined and manipulated with the full facilities of OBJ3's higher-order module system. Theories can be extended and/or joined, abstracted, instantiated, and have multiple views. OBJ3's open and close mechanisms allow extending modules with equations for theorems proved within them. It is also easy to construct theories that are abstract over one or more theories. For example, proofs in group theory could be constructed in the module

```
obj GROUP_PFS[ G :: GROUP ] is endo
openr GROUP_FPS .
*** Construct group theory proofs here
close
```

An instance of the resulting module could then be constructed using a command of the form

```
make GND_GROUP_PFS is GROUP_PFS[GroupMod] endm
```

where GroupMod is a view of some OBJ3 module that satisfies GROUP. The resulting module GND_GROUP_PFS would contain concrete instances of the proofs constructed in GROUP_PFS. All these mechanisms also apply to tactics, because tactics are expressed in OBJ3. Thus we do not need separate mechanisms to manage how theorems depend on the tactics used to prove them.

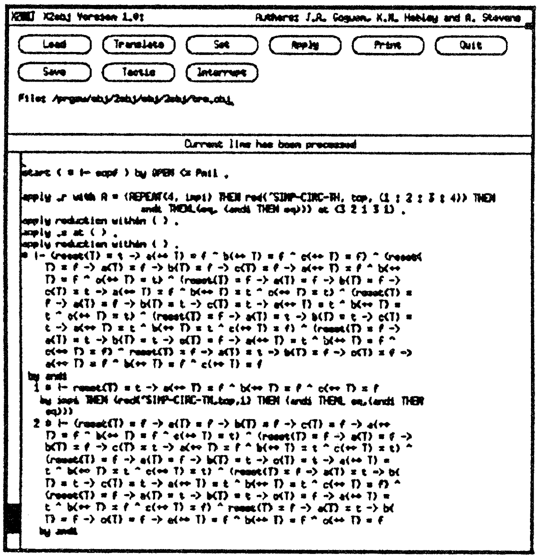
(*e*) *2OBJ as a hybrid framework*

A key practical drawback of tactic based mechanized theorem proving is its relative inefficiency compared to term rewriting and custom-built decision procedures, because of the overhead in explicitly invoking inference rules. Implementing such procedures from scratch can be expensive, but 2OBJ can use the OBJ3 interpreter for controlled rewriting of object logic terms in named OBJ3 modules, and can also call procedures coded in the underlying LISP.

The combination of OBJ3's interpreter for efficient rewriting modulo associativity, commutativity, and/or identity, and its mechanisms for controlling the rules used for rewriting, makes it possible to implement rewriting systems for specific proof tasks very efficiently. We intend to extend the semantics outlined in §5 to cover automatic compilation of suitable axioms and lemmas in the object logic into equivalent OBJ3 equations. For example, the axiom

```
|A| x . |A| y . |A| z. append(cons(x,y),z) = cons(x,append(y,z))
```

Figure 1



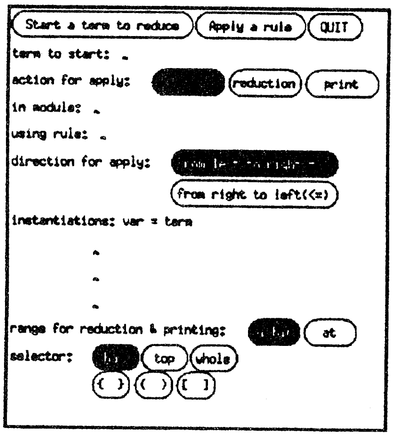
```
File: /prgms/obj/2obj/obj/2obj/bra.obj

Current line has been processed

start ( @ |- exp# ) by OPEN Cx Fail .

apply .r with A = (REPEAT(4, imp1) THEN red('SIMP-CIRC-TH, top, (1 : 2 : 3 : 4)) THEN
                    andi THENL(eq, (andi THEN eq))) at (3 2 1 3 1) .
apply reduction within ( ) .
apply .s at ( ) .
apply reduction within ( ) .
@ |- (reset(T) = t -> a(++ T) = f ^ b(++ T) = f ^ c(++ T) = f) ^ (reset(
    T) = f -> a(T) = f -> b(T) = f -> c(T) = f -> a(++ T) = f ^ b(++
    T) = f ^ c(++ T) = t) ^ (reset(T) = f -> a(T) = f -> b(T) = f ->
    c(T) = t -> a(++ T) = f ^ b(++ T) = t ^ c(++ T) = t) ^ (reset(T) =
    f -> a(T) = f -> b(T) = t -> c(T) = t -> a(++ T) = t ^ b(++ T) =
    t ^ c(++ T) = t) ^ (reset(T) = f -> a(T) = t -> b(T) = t -> c(T) =
    t -> a(++ T) = t ^ b(++ T) = t ^ c(++ T) = f) ^ (reset(T) = f ->
    a(T) = t -> b(T) = t -> c(T) = f -> a(++ T) = t ^ b(++ T) = f ^
    c(++ T) = f) ^ reset(T) = f -> a(T) = t -> b(T) = f -> c(T) = f ->
    a(++ T) = f ^ b(++ T) = f ^ c(++ T) = f
 by andi
  1 @ |- reset(T) = t -> a(++ T) = f ^ b(++ T) = f ^ c(++ T) = f
   by imp1 THEN (red('SIMP-CIRC-TH,top,1) THEN (andi THENL eq,(andi THEN
    eq)))
  2 @ |- (reset(T) = f -> a(T) = f -> b(T) = f -> c(T) = f -> a(++
    T) = f ^ b(++ T) = f ^ c(++ T) = t) ^ (reset(T) = f -> a(T) = f ->
    b(T) = f -> c(T) = t -> a(++ T) = f ^ b(++ T) = t ^ c(++ T) = t) ^
    (reset(T) = f -> a(T) = f -> b(T) = t -> c(T) = t -> a(++ T) =
    t ^ b(++ T) = t ^ c(++ T) = t) ^ (reset(T) = f -> a(T) = t -> b(
    T) = t -> c(T) = t -> a(++ T) = t ^ b(++ T) = t ^ c(++ T) = f) ^
    (reset(T) = f -> a(T) = t -> b(T) = t -> c(T) = f -> a(++ T) =
    t ^ b(++ T) = f ^ c(++ T) = f) ^ reset(T) = f -> a(T) = t -> b(
    T) = f -> c(T) = f -> a(++ T) = f ^ b(++ T) = f ^ c(++ T) = f
   by andi
```

Figure 2



```
term to start: _
action for apply:
in module: _
using rule: _
direction for apply:
instantiations: var = term
range for reduction & printing:
selector:
```

Figure 1. Window layout for 2OBJ tool.
Figure 2. The apply popup.

expressed in FOPC_SENTENCE syntax should translate to the OBJ equation

```
eq append(cons(x,y),z) = cons(x,append(y,z)) .
```

Such a compiler would make 2OBJ a hybrid logical framework, in which some object logic proofs are internalized as OBJ3 rewriting. 2OBJ can already apply manually supplied internal axioms.

Some industrial applications require special purpose programs in a low level imperative language; a good example is the boolean decision diagrams (BDDS) used in hardware verification (Hoereth 1991). 2OBJ tactics can invoke such efficient proof procedures by using OBJ3's 'built-in' facility to attach LISP and/or C code to OBJ3 operations.

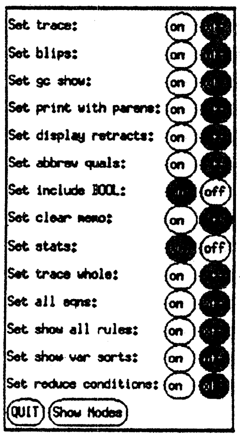
5. The 2OBJ tool

The 2OBJ tool is a customizable X11 window harness for 2OBJ, with three main components: a command panel showing the available options; a small area for system messages; a text window for proof editing and interaction with OBJ3.

The text field provides in-line editing of OBJ3 command lines, in an Emacs-like shell, with user definable key bindings (see figures 1 and 2).

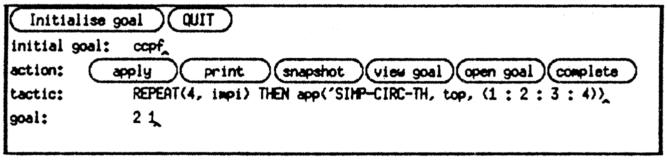
The command panel has nine buttons and a text field labelled 'File:' for giving file-names to be loaded or saved. The buttons either provide an operation (e.g. the *print* button sends a screendump to the printer), or an auxiliary operation via a popup. The apply popup gives a user interface for OBJ3's *apply* operation, which applies a given rule from a named module to a particular position of a given term. Variable instantiation fields (which are needed for applying some rules in the reverse direction) can be dynamically added. The *set* popup provides a sheet for setting various OBJ3 system parameters. The *tactic* popup provides a user interface for applying proof tactics to goals and subgoals. The user need only give goal descriptions or tactics, as the harness provides tactic language syntax. Unedited text

Figure 3



Set trace: on [✓ off]
Set blips: on [✓ off]
Set gc show: on [✓ off]
Set print with parens: on [✓ off]
Set display retracts: on [✓ off]
Set abbrev quals: on [✓ off]
Set include BOOL: [✓ on] off
Set clear memo: on [✓ off]
Set stats: [✓ on] off
Set trace whole: on [✓ off]
Set all eqns: on [✓ off]
Set show all rules: on [✓ off]
Set show var sorts: on [✓ off]
Set reduce conditions: on [✓ off]

Figure 4



```
initial goal:   ccpf
tactic:         REPEAT(4, impi) THEN app('SIMP-CIRC-TH, top, (1 : 2 : 3 : 4))
goal:           2 1
```

Figure 3. The set popup.

Figure 4. The tactic popup.

fields and buttons remain unaltered for use in subsequent rule applications. Feedback is manipulated before being displayed to provide more informative descriptions. (See figures 3 and 4.)

6. Foundations

A project like 2OBJ raises difficult issues about foundations, including: What is logic? What is encoding? What does it mean for an encoding to be correct? This section informally describes some categorical metamathematics to support using equational logic as a logical framework; details will appear elsewhere. We enrich parchments, charters and institutions (as in Goguen & Burstall (1984, 1986)) to include proofs, and we define notions of encoding and correctness at this level of abstraction.

(*a*) *Institutions, charters and parchments*

The theory of institutions formalizes the intuitive notion of logical system (Goguen & Burstall 1984). An *institution* consists of: a category of signatures (giving non-logical symbols); a set of Σ-sentences for each signature Σ; a category of Σ-models for each Σ; and a Σ-satisfaction relation between Σ-sentences and Σ-models for each Σ. The basic axiom says that truth (i.e. satisfaction) is invariant under change of notation (by signature morphisms).

To ease the construction of institutions, Goguen & Burstall (1986) introduced charters and parchments. A *charter* consists of: a category of signatures; a category of 'syntactic' systems; an adjunction between these two categories (a categorical version of the fact that the syntax of a logical system is freely generated over its signature); a set-valued 'base' functor which extracts the sentences from these syntactic systems; and a 'ground' object which is mapped by the base functor to $\{true, false\}$, and used for interpreting syntactic systems as models.

Although satisfaction may be obtained automatically by chartering, it is still some trouble to construct the category of syntactic systems. Parchments give these for free. A *parchment* consists of: a category of signatures; an equational signature Σ^+ for every signature Σ, giving syntax; a sort '$*$' such that $(T_{\Sigma^+})_*$ is the set of all Σ-sentences; a 'ground' signature Γ; and a Γ^+-algebra in which to interpret sentences. Given a parchment, the syntactic systems of the charter to which it gives rise are pairs whose first component is a signature, and whose second component is the initial algebra of the corresponding equational signature.

(b) *Logics as ruled institutions and parchments*

To define what it means to encode an object logic into order sorted equational logic, we will equip institutions with rules of inference and the proofs they generate, and we will assume that the object logic is written on parchment, i.e. that its syntax is an initial algebra.

More precisely, a *ruled institution* consists of: a category of signatures; a category of sets of Σ-sentences with Σ-proofs as morphisms between them, generated by the Σ-rules of inference for each Σ, plus the inclusions, and closed under products; and for each Σ, a Σ-satisfaction relation between Σ-sentences and proofs, and Σ-models, such that rules of inference, inclusions, products, and satisfaction are preserved by signature morphisms. Ruled institutions may be constructed from *ruled parchments*, which have: an equational signature Σ^{+} for every signature Σ, providing the syntax; a sort '$*$' such that $(T_{\Sigma^{+}})_{*}$ is the set of all Σ-sentences; a sort '$\mapsto$' such that $\Sigma^{+}_{\mapsto}$ contains precisely the Σ-rules of inference; a 'ground' signature Γ; and a Γ^{+}-algebra in which to interpret sentences and proofs.

We can now rephrase and clarify our assumptions by saying that object logics are written on ruled parchment, and hereafter we assume that every object logic is given in this way. It may be surprising that ruled parchments (and ruled institutions) are automatically sound. In fact, soundness is implicit in $(\_)^{+}$ being a functor from the category of object logic signatures to the category of equational signatures.

(c) *Encoding object logics into equational logic*

To encode an object logic $\mathscr{L}$ into equational logic (hereafter denoted $\mathscr{E}\mathscr{Q}$), for every $\mathscr{L}$ signature Σ, we need an equational theory presentation $\widehat{\Sigma^{+}}$ with a sort '$*$' such that there is a bijection (natural in Σ) between the set $(T_{\Sigma^{+}})_{*}$ of all $\mathscr{L}$, Σ-sentences, and the set $(T_{\Sigma^{+}})_{*}$ of all terms that encode sentences; i.e. an encoding should provide a distinct representation for each object logic sentence. Given a Σ-sentence e, let us denote the term encoding it by $\hat{e}$. Next, the encoding of an $\mathscr{L}$ rule of inference R is a set $\hat{R}$ of rewrite rules such that for any set S of sentences, $\hat{R}$ rewrites $\hat{S}$ to $\hat{S}'$ iff $S \rightarrow S'$ is an instance of R, and $\hat{R}$ leaves $\hat{S}$ unchanged iff R does not apply to S, so that deduction on encoded sentences exactly mirrors deduction in the object logic. (Actually, S and S' above may be any *aggregate* of sentences, such as multisets or lists, having a predicate '$\subset$' for aggregate inclusion, and an operation '$\cup$' for combining aggregates. Note that we have extended '$\hat{}$' to aggregates of sentences.)

The above encoding of the inference rule R was typical of an *indirect* encoding. In the case of a *direct* encoding, all the sentences are *equations*, and all the rules are equational (see next section). However, many logics have a non-trivial 'intersection' with $\mathscr{E}\mathscr{Q}$; some sentences are equations, and some of the rules are equational. It follows that when reasoning in an encoding of an object logic theory, we shall sometimes want to apply indirectly encoded rules (for the non-equational steps), and OBJ rewriting at other times (for the equational ones). This 'dual' approach to deduction can of course only work if the two terms that make up an equation term (the encoding) are the same as the two terms that make up (an instance of) the $\mathscr{E}\mathscr{Q}$ equation. In other words, the terms which denote terms in the object logic signature have to be the same as the terms which they denote. Such an encoding we refer to as a *hybrid* encoding. Clearly hybrid encodings are perfectly legitimate with respect to our notion of encoding, and hence the proposition below applies to them also.

The set of Σ *provability terms* contains all terms $\hat{A} \vdash_\Sigma \hat{C}$ such that $A \rightarrow C$ is an instance of an inference rule, or else $C \subseteq A$ satisfying closed under the operations *combine* and *compose*, and the equations

$$\begin{aligned} &\mathit{compose}\ \{\Phi \vdash_\Sigma \Phi', \Phi' \vdash_\Sigma \Phi''\} = \Phi \vdash_\Sigma \Phi'', \\ &\mathit{combine}\ \{\Phi \vdash_\Sigma \Phi', \Phi \vdash_\Sigma \Phi''\} = \Phi \vdash_\Sigma \Phi' \cup \Phi''. \end{aligned}$$

(The notation '$\vdash_\Sigma$' used here should not be confused with the sequents used in sequent presentations of logic.) We then have the following:

Proposition 1. *The bijection '$\hat{\ }$' extends to a bijection (natural in Σ) between the set of Σ provability terms and PO ($\mathscr{P}Sen_{\mathscr{L}}(\Sigma)$), where $\mathscr{P}Sen_{\mathscr{L}}(\Sigma)$ is the category of all Σ proofs in $\mathscr{L}$, and PO is the functor which maps a category to the preorder obtained by identifying all arrows having the same source and target.*

(d) *Adequacy and faithfulness*

An important question for any logical framework is how proof terms in the encoded object logic relate to proofs in the object logic. Let us call an encoding of a logic $\mathscr{L}$ in 2OBJ a *frame*, and denote it 2OBJ[$\mathscr{L}$]. Then a frame 2OBJ[$\mathscr{L}$] is *adequate* iff it can prove everything $\mathscr{L}$ can, and is *faithful* iff it can prove no more than that. The following is immediate from Proposition 1 and our general assumptions:

Theorem 2. 2OBJ[$\mathscr{L}$] *is adequate and faithful.*

(e) *Internal encoding*

Many logics $\mathscr{L}$ have a subset of their syntax which is essentially equational logic. For example, in first-order logic with equality, a sentence $(\forall n, m\colon Nat) = (n+m, m+n)$ corresponds to the equation $(\forall n, m\colon Nat)\ n+m = m+n$. Also, any first-order theory consisting of such equational sentences translates into equational logic, where one can deduce consequences, and then translate them back. This process is faithful in the sense that any such derivation has a corresponding proof entirely in first-order logic with equality.

We can formalize and generalize this to any framework logic $\mathscr{F}$ and object logic $\mathscr{O}$ as follows: *a ruled institution morphism* from $\mathscr{O}$ to $\mathscr{F}$ consists of a functor E from the category of object logic signatures to the category of framework logic signatures, and for every $\mathscr{O}$ signature Σ, a subset $F_{\mathscr{O}}\Sigma$ of $Sen_{\mathscr{O}}\Sigma$ plus functions $o_\Sigma\colon F_{\mathscr{O}}\Sigma \rightarrow Sen_{\mathscr{F}} E\Sigma$ and $m_\Sigma\colon Sen_{\mathscr{F}} E\Sigma \rightarrow \varphi_{\mathscr{O}}\Sigma$ that are order-preserving (on the underlying preorders induced by identifying all arrows having the same source and target in the respective categories of proofs) and form a Galois connection from $F_{\mathscr{O}}\Sigma$ to $(Sen_{\mathscr{F}} E\Sigma)^{op}$. The function o maps sentences to the object level, while the function m maps sentences to the meta level. This is the kind of encoding used by logical (as opposed to *meta*-logical) frameworks. From Proposition 1 we now obtain

Proposition 3. *Internal encoding preserves adequacy and faithfulness.*

This helps explain how deduction in a frame 2OBJ[$\mathscr{L}$] can proceed at both the meta and the object level.

(f) *Soundness of labelled proof trees*

The implementation of proof trees in 2OBJ is actually a bit more complex than indicated in §5*c*, because it can record individual rule applications. We now briefly describe an algebra of labelled proof trees, and argue that its operations are sound.

If Σ is an object logic signature, then *labelled* Σ-*proof trees* are specified as follows: a label is an inference rule, or *open*, or *closed*; leaves are pairs $(S, \textit{open})$ or $(S, \textit{closed})$, where S is a set of Σ-sentences; nodes are pairs (S, R) where S is a set of Σ-sentences and R is an inference rule such that $R\colon \cup_i S_i \mapsto S$ where the S_i are the first coordinates of the predecessors of (S, R); if P a proof tree with its list $[G_i]$ of open subgoals, and if $[P_i]$ is a list of proofs such that $\textit{goal}(P_i) = G_i$ for each i, then their composition is the tree resulting when the G_i in P are replaced by the P_i; if P is a proof tree, then the removal of all internal nodes, and the replacement of the label of the goal of P by '?' (to indicate loss of proof structure) yields a tree denoted $\textit{fold}(P)$; the leaves $(\emptyset, l)$ of a proof tree, where l is a label, may be replaced by $(\emptyset, \textit{closed})$. The soundness of this representation is stated as follows:

Proposition 4. *P is a proof tree iff there is a Σ-proof*: $\cup\, \textit{subgoals}(\textit{fold}(P)) \rightarrow \textit{goal}(P)$.

The proof is by induction on the structure of P. Note that sequents of the form $S \vdash_\Sigma S'$ from §5*c* can be identified with proof trees $[(S', ?), (S, \textit{open})]$, for $S' \subseteq S$.

(*g*) *A frame for first-order logic*

This subsection very briefly sketches a ruled parchment presentation and encoding into $\mathscr{E}\mathscr{Q}$ for many sorted first order logic, henceforth written $\mathscr{FOL}$. Given an $\mathscr{FOL}$-signature (S, Σ, Π), then $(T_{(S,\Sigma,\Pi)^+})_*$ contains sequents of the form $H \vdash X$, where H is a set of Σ-sentences and X is a Σ-sentence (the sequents are therefore the 'sentences' in this presentation). $(T_{(S,\Sigma,\Pi)^+})_{\mapsto}$ contains the instances of the $\mathscr{FOL}$-rules of inference; for example it contains $\{H \vdash X, H \vdash Y\} \rightarrow \{H \vdash X \wedge Y\}$ whenever H is a set of Σ-sentences and X, Y are Σ-sentences, corresponding to the rule *andi*. The presentation of the other rules is similar. The encoded proof trees are constructed following the general recipe given above. For example, an application of *andi* in a proof tree corresponds to a subtree of the form

$$[(\{H \vdash X \wedge Y\}, \textit{andi}), [(\{H \vdash X\}, l_1), (\{H \vdash Y\}, l_2)]].$$

7. Future research

Our research into hybrid logical frameworks and the 2OBJ architecture is still at a relatively early stage. Besides further work on specific applications of 2OBJ (especially hardware verification), research topics still to be addressed include the following.

1. *Induction principles for order sorted algebra.* The current version of 2OBJ lacks a built in induction principle. It is the responsibility of the user to provide appropriate inference rules in a given frame. The FOPC frame, for example, provides a simple mechanism by which the user can enumerate a set of constructors for a given sort. OBJ3 itself does not provide induction principles. It is, therefore, a research priority to provide a mechanism for the automatic derivation of structural induction principles from data constraints (in the sense of Goguen & Burstall (1984)) on ADTS defined in OBJ3; we view data constraints as providing the specification side of induction.

2. *Mechanized derivation of OBJ equations from (encoded) theorems.* As mentioned in §3(*e*), we wish to develop a mechanism to translate suitable (encoded) theorems into equivalent OBJ3 equations.

3. *Support for the proof and use of meta-theorems.* 2OBJ currently provides very spartan facilities for constructing meta-level proofs. Much more extensive machinery will be needed to make meta-level reasoning like that mentioned in §1*a* workable in practice.

4. *Proof refinement.* A common complaint about current mechanized proof systems is that they are inconvenient for developing proofs that are not already fully worked out, as these systems only support a rather awkward bottom-up approach to proof construction, rather than a more natural top-down approach in which proofs are first constructed as skeletons, and then may refined into full proofs. Because 2OBJ embeds its tactic language and its ADT of object logic proofs in its framework logic, it is possible to reason about proofs in 2OBJ by reasoning about tactics that construct them. Hence, proof skeletons can be represented in 2OBJ as partial, non-executable, specifications for proof tactics. Thus, by providing mechanisms for constructing and refining tactic specifications, 2OBJ could support the construction and refinement of proof skeletons. A more short-term approach is to allow subproof-valued variables in proofs.

5. *Frame compiler.* We intend to build a compiler from ruled parchments to 2OBJ frames, where ruled charter specifications are expressed in OBJ3 itself, and each rule is mapped to a 'button' in the user interface.

We thank Mr Tim Winkler for help with details of the OBJ3 system upon which 2OBJ is built, and we also thank him and all others who contributed to the construction of OBJ3, including Dr José Meseguer, Dr Kokichi Futatsugi, Dr Claude Kirchner, Dr Hélène Kirchner, and Professor Jean-Pierre Jouannaud. Production of the 2OBJ tool has been aided by some earlier work of Mr Han Yan on a tool for OBJ3, and by software distributed by the National Semiconductor Corporation. The exposition of this paper has benefited greatly from terminology introduced in Constable & Basin (1991). The research reported in this paper has been supported in part by grants from the SERC, the Department of Trade and Industry, and the Fujitsu Corporation.

References

Boyer, R. & Moore, J. 1980 *A computational logic*. Academic.

Bundy, A., van Harmelen, F., Horn, C. & Smaill, A. 1990 The Oyster–Clam system. In *10th International Conference on Automated Deduction* (ed. M. E. Stickel), pp. 647–648. Lecture Notes in Artificial Intelligence no. 449. Springer-Verlag.

Constable, R. & Basin, D. 1991 Meta-logical frameworks. In *Proceedings 2nd Edinburgh Workshop on Logical Frameworks*. Cambridge University Press.

Constable, R. *et al.* 1986 *Implementing mathematics with the NuPRL proof development system*. Prentice-Hall.

Futatsugi, K., Goguen, J., Jouannaud, J. & Meseguer, J. 1985 Principles of OBJ2. In *Proceedings Twelfth ACM Symposium on Principles of Programming Languages* (ed. B. Reid), pp. 52–66. Association for Computing Machinery.

Futatsugi, K., Goguen, J., Meseguer, J. & Okada, K. 1987 Parameterized programming in OBJ2. In *Proceedings Ninth International Conference on Software Engineering* (ed. R. Balzer), pp. 51–60. IEEE Computer Society.

Goguen, J. 1990 Higher-order functions considered unnecessary for higher-order programming. In *Research Topics in Functional Programming* (ed. D. Turner), pp. 309–352. Addison-Wesley. University of Texas at Austin Year of Programming Series; preliminary version in SRI Technical Report SRI-CSL-88-1.

Goguen, J. & Burstall, R. 1986 A study in the foundations of programming methodology: specifications, institutions, charters and parchments. In *Proceedings Conference on Category Theory and Computer Programming* (ed. D. Pitt, S. Abramsky, A. Poigné & D. Rydeheard), pp.

313–333. Lecture Notes in Computer Science, Volume 240; also, Report Number CSLI-86-54, Center for the Study of La.guage and Information, Stanford University.

Goguen, J. & Burstall, R. 1992 Institutions: abstract model theory for specification and programming. *J. Assoc. Comput. Machinery*. Draft, as Report ECS-LFCS-90-106, Computer Science Department, University of Edinburgh, January 1990; an early ancestor is Introducing institutions, in *Proceedings Logics of Programming Workshop* (ed. E. Clarke & D. Kozen), pp. 221–256. 1984 Springer Lecture Notes in Computer Science, vol. 164.

Goguen, J., Winkler, T., Meseguer, J., Futatsugi, K. & Jouannaud, J. 1992 Introducing OBJ. In *Applications of algebraic specification using OBJ* (ed. J. Goguen, D. Coleman & R. Gallimore). Cambridge.

Gordon, M. 1985 HOL: a machine oriented formulation of higher-order logic. *Tech. Rep.* 85. University of Cambridge, Computer Laboratory.

Gordon, M., Milner, R. & Wadsworth, C. 1979 *Edinburgh LCF*. Springer. Lecture Notes in Computer Science, vol. 78.

Harper, R., Honsell, F. & Plotkin, G. 1987 A framework for defining logics. In *Proceedings Second Symposium on Logic in Computer Science*, pp. 194–204. IEEE Computer Society.

Harper, R., Sannella, D. & Tarlecki, A. 1987 Structure and representation in LF. *Tech. Rep.* ECS-LFCS-89-75. Laboratory for Computer Science, University of Edinburgh.

Hoereth, S. 1991 Improving the performance of a BBD-based tautology checker. In *Proceedings Advanced Research Workshop on Correct Hardware Design Methodologies*. (In the press.)

Matthews, S., Smaill, A. & Basin, D. 1991 Experience with FS0 as a framework theory. In *Proceedings 2nd Edinburgh Workshop on Logical Frameworks*. Cambridge University Press.

McCune, W. 1989 The Otter user's guide. *Tech. rep.* Argonne National Laboratory Technical Report.

Miller, D. 1990 A logic programming language with lambda-abstraction, and simple unification. *Extensions of logic programming*, (ed. P. Shroeder-Heister). Springer, Lecture Notes in Artificial Intelligence.

Paulson, L. C. 1987 *Isabelle: the next 700 theorem provers. In Proceedings Second Symposium on Logic in Computer Science*, pp. 194–204. IEEE Computer Society.

Paulson, L. C. 1988 The foundation of a generic theorem prover. *Tech. Rep.* 130. University of Cambridge, Computer Laboratory.

Sawamura, H., Minami, T., Yokota, K. & Ohashi, K. 1990 A logic programming approach to specifying logics and constructing proofs. In *Proceedings of 7th International Conference on Logic Programming* (ed. D. H. D. Warren & P. Szeredi), pp. 405–424. MIT Press.

Discussion

W. A. HUNT (*Computational Logic, Inc., U.S.A.*). Can a database facility be implemented for a logic encoded in 2OBJ? Can proofs be looked up?

J. A. GOGUEN. The underlying OBJ3 system already provides a very flexible database which can store (parametrized) theories. Such theories could certainly contain proof terms. We do not currently provide any special support for database search, but I expect that something useful and interesting could be done based on the existing default view mechanism of OBJ3.

T. F. MELHAM (*University of Cambridge, U.K.*). What are the prospects for using this system for meta-theory? In particular, will the fact that everything is based on equational logic restrict the kinds of meta-theoretic reasoning that can be done?

J. A. GOGUEN. Not only formulae, but also proofs, are encoded as terms; indeed, it is one of the strengths of our approach is that it is straightforward to reason about proofs; there are no restrictions. (See the paper for more details.)