Bourbaki Proof Checker

Juha Arpiainen

October 20, 2005

Contents

1 Introduction

Bourbaki is a program for writing and verifying formal mathematical proofs. It is mainly based on Norman Megill's *Metamath*, but aims on providing more powerful syntax and tools for automating proof writing. Performance is not a design goal, at least as long as writing proofs takes longer than verifying them.

Nicolas Bourbaki is the pseudonym of a group of mathematicians, who have written a rigorous multi-volume treatise of Mathematics based on set theory. My long term goal with Bourbaki is likewise a rigorous on-line computer-verified dictionary of Mathematics.

The current version 3.5 is implemented as an embedded language on top of ANSI Common Lisp (the version numbering starts from 3 in memory of the time wasted in two unsuccesful C++ versions). This means most of Lisp features are available for the user, in particular it is possible to write Lisp functions that construct proofs (See section 7).

1.1 Quick start

```
$ gunzip bourbaki.tar.gz
$ tar xvf bourbaki.tar

# Bourbaki uses UTF-8 encoding
# Substitute clisp with your favourite Common Lisp implementation
$ LC_ALL=en_US.utf8 clisp

> (load "init")
> (in-package bourbaki-user)

;; load file "logic.lisp" and verify all theorems
;; defined there
> (verify !!logic)

;; show theorem "id"
> (print-theorem !logic!id)
```

2 Symbols

Mathematical expressions in Bourbaki are trees of symbols written like Lisp s-expressions. Thus we write $[\rightarrow\ [\wedge\ \phi\ \psi]\ \chi]$ instead of $(\phi\ \wedge\ \psi)\ \rightarrow\ \chi$. As all symbols in Bourbaki have a definite number of arguments, the inner parentheses in the first expression are redundant; it is equal to $[\rightarrow\ \wedge\ \phi\ \psi\ \chi]$.

Each symbol in Bourbaki has a name, a type and a list of argument types. Primitive symbols can be defined with macro 'prim':

```
;; ASCII tokens are used here instead of the actual
;; Unicode mathematical symbols in prop.lisp

;; Symbol "->" is of type pr and has two arguments
;; of type pr
(prim pr "->" pr pr)
(prim pr "\forall" sv pr)
```

Here pr (for 'proposition' or 'predicate') is a symbol type defined with 'make-bsymtype':

```
(setq pr (make-bsymtype :eq-op nil :is-bound nil :super nil))
```

A symbol type can be defined as a subtype of another type by giving the :super argument to make-bsymtype. For the other arguments of make-bsymtype, see section 5. Usually one symbol type ('pr' in the Bourbaki Libary) would correspond to the logical statements of the formal system.

The symbols are stored in a namespace (actually a hierarchy of namespaces, see section 6) separate from the Lisp package system. Symbol types on the other hand are stored as values of normal Lisp symbols. The name of a symbol may contain any Unicode characters.

3 Expressions

An expression, or *symtree* T is stored internally as a list of the form $(op\ tree_1\ tree_2\ \ldots\ tree_n)$ where *op* is a symbol (the *operator* of the tree) and each $tree_i$ is a symtree ($n = 0$ for the leaves of the tree). The *type* of T is the type of *op*.

The tree is *well-formed* if the number of arguments of *op* is n and each $tree_i$ is a well-formed tree whose type is a *subtype* of the corresponding argument type of *op*.

3.1 Functions related to symtrees

The type of a symbol or symtree can be found with the function 'wff-type'. Well-formedness of a symtree can be tested with the function 'proper-wff-p' (function 'wffp' returns also true for the empty list). The equality of two symtrees can be tested with the function 'wff-equal'. (The word wff for 'well formed formula' is normally used only for well-formed symtrees with type 'pr'. proper-wff-p tests symtrees of any type, however).

Symtrees are normally created with the [ ] syntax: [$\rightarrow\ \phi\ \psi$] returns the tree (*imp* (*psi*) (*phi*)), where *imp*, *phi* and *psi* are the Lisp objects corresponding to symbols $\rightarrow$, ϕ and ψ. In this document the [ ] syntax is sometimes used to represent the actual tree although this is not exactly correct. The tree might also be shown as ($\rightarrow$ (ψ) (ϕ)). Note that the arguments are internally stored in *reversed* order for technical reasons.

[is actually a reader macro that gets converted to a call to 'symstr-parse'. symstr-parse would normally not be called directly.

It is often necessary to substitute certain symbols in a symtree with other symtrees. For example, substituting [ψ] for ϕ and [$\rightarrow\ \phi\ \phi$] for ψ in [$\rightarrow\ \phi\ \rightarrow\ \psi\ \phi$] results in [$\rightarrow\ \psi\ \rightarrow\ \rightarrow\ \phi\ \phi\ \psi$]. The Common Lisp function 'sublis' almost does this: The result would be ($\rightarrow$ ($\rightarrow$ ((ψ)) (($\rightarrow$ ϕ ϕ))) ((ψ))) with extraneous pairs of parenthesis. The function 'replace-vars-canonize' is a wrapper around sublis that makes the result well-formed. It is called with two arguments: the original symtree and a list of pairs (sym_i . $subst_i$) of substitutions.

Symtrees are printed with the function 'print-symtree'. Its operation depends on value of the variable *bourbaki-debug*. If set to true, the internal list structure is printed, otherwise the [ ] format is used. print-symtree also takes an optional boolean argument 'paren' (default value nil). Using this when not in debug mode results in additional parentheses being printed for clarity.

4 Theorems

Theorems in bourbaki have zero or more *variables* (or *arguments*), an *assertion*, zero or more *hypotheses* and a *proof*. The theorem is treated as a transformation rule: when its hypotheses are satisfied for some symtrees substituted for its variables, its (correspondingly substituted) assertion can be used to prove other theorems.

Theorems are created with the macros ‘th’, ‘ass’, ‘hypo’ and ‘proof’. Example corresponding to *Metamath* theorem ”id1”:

```
;; Theorem "id" has one variable of type pr
(th "id" (pr "phi")
    (hypo) ; No hypotheses, this line could be omitted
    (ass [-> phi phi])
    (proof
      [ax1 [phi] [phi]] ; or equally [ax1 phi phi]
      [ax1 [phi] [-> phi phi]]
      [ax2 [phi] [-> phi phi] [phi]]
      [ax-mp [-> phi -> -> phi phi phi]
             [-> -> phi -> phi phi -> phi phi]]
      [ax-mp [-> phi -> phi phi] [-> phi phi]]))
```

Theorems share the same namespace with symbols, there cannot be a symbol ”id” and a theorem ”id” at the same time. The proof contains theorem references (that have same syntax as symtrees) with explicit substitutions for the referred theorems’ variables, as opposed to the implicit substitutions *Metamath* uses.

Theorems can be printed with the function ‘print-theorem’

4.1 Theorem syntax

Syntax for the macros ‘th’, ‘ass’, ‘hypo’ and ‘proof’ follows.

4.1.1 th

```
theorem-form := ( th name var-spec lisp-form* )
name         := string
var-spec     := ( { type var-names }* )
type         := lisp-form
var-names    := string | ( string* )
```

Multiple variables of same type can be declared as *type* (”var_1” ”var_2” … ”var_n”) instead of *type* ”var_1” *type*”var_2” … *type* ”var_n”. Evaluating each type should result in an object of type ‘bsymtype’.

4.1.2 hypo

```
hypothesis-form := ( hypo symtree* )
```

Multiple hypothesis forms may be used; the order does not matter.

4.1.3 ass

```
assertion-form := ( ass symtree )
```

Each use of 'ass' *replaces* the previous assertion. The type of each hypothesis and assertion would normally be 'pr'; however, Bourbaki allows any type to be used. If all the axioms have assertions of type 'pr', theorems of no other type can be verified.

4.1.4 proof

```
proof-form := ( proof symtree* )
```

The operator of each proof line should be an already proved theorem, an axiom or a definition (see section DEF).

Using multiple proof forms has somewhat unexpected results: New lines are added at the *beginning* of the proof.

```
(proof A B C)
(proof D E F)
```

has the same effect as

```
(proof D E F A B C)
```

The relative order of hypo-, ass- and proof-forms does not matter.

4.2 Axioms

Axioms are defined similarly, but using 'ax' instead of 'th'. Bourbaki allows using 'proof' for axioms even though having a proof for an axiom does not make much sense.

4.3 Verification

The function 'verify' checks the correctness of a theorem or an axiom. The verification algorithm for a theorem *thr* is as follows:

1. Check that the hypotheses and assertion of *thr* are well-formed.
2. Initialize list L with the hypotheses of *thr*.
3. For each proof line [$ref\ subst_1\ \ldots\ subst_n$],
    - Verify *ref* if it is not already verified.
    - Check that the number of variables of *ref* is equal to the number n of substitutions, and that each $subst_i$ is well-formed and of correct type.
    - For each hypothesis h of *ref*, check that (replace-vars-canonize h ((var_1 . $subst_1$) ...)) is in L.
    - Insert (replace-vars-canonize a ((var_1 . $subst_1$) ...)) into L, where a is the assertion of *ref*.
4. Check that the result of the last proof line is equal to the assertion of *thr*.

The relative order of independent proof lines does not matter. For example, the first three lines of the proof of "id" could have been in any other order.

4.4 Distinct Variables

Distinct variable conditions are introduced with the macro 'dist'. This is equivalent to *Metamath's* $d feature, see the Metamath book[1] for details. Distinct variable syntax:

```
distinct-variable-form := ( dist condition* )
condition              := ( symbol* )
```

5 Definitions

New symbols can be defined in terms of primitive (or already defined) symbols with the 'def' macro. The *soundness* of Bourbaki definitions is carefully checked. A symbol s introduced with the sound definition d can be *eliminated*, that is, each wff containing s can be proved, using only def, other sound definitions, and axioms, to be equivalent with a wff containing only primitive symbols. Also, each theorem thr whose proof uses def, could be proved without using def, provided that s does not occur in the hypotheses or the assertion of thr.

def allows a limited kind of definitions, whose soundness can be proved. The syntax for definitions is as follows:

```
definition-form := ( def def-name sym-type sym-name sym-vars dummy-vars rhs )
def-name        := string
sym-type        := lisp-form
sym-name        := string
sym-vars        := var-spec
dummy-vars      := var-spec
rhs             := symtree
```

A definition (def "df-sym" t_0 "sym" (t_1 "var_1" ...) (u_1 "$dummy_1$" ...) rhs) is equivalent with

```
(prim t_0 "sym" t_1 ... t_n)
(ax "df-sym" (t_1 "var_1" ... t_n "var_n" u_1 "dummy_1" ... u_m "dummy_m")
    (ass [eq [sym t_1 ... t_n] rhs])),
```

provided it satisfies the soundness test. Here 'eq' is the *equality operator* for type t_0. eq should be an *equivalence relation* and have the *substitution property* for objects of type t_0: Each wff of the form [$\rightarrow$ [$eq\ x\ y$] [$\leftrightarrow$ $\phi(x)$ $\phi(y)$]] should be provable. These properties are not checked by Bourbaki; using a bad equality operator will lead to incorrect definitions.

The equality operator of a type can be set with (setf (bsymtype-eq-op t_0) eq). Note that the equality operator "$\leftrightarrow$" for type 'pr' cannot be defined with def. The equivalent of *Metamath* df-bi must be an axiom in Bourbaki.

5.1 Free and Bound Symbols

In addition to the variables "var_i" the right-hand side of a definition may contain *dummy variables* (the $dummy_i$). These must only occur *bound* on the rhs.

In Bourbaki a symbol type may be declared bound with the :is-bound argument to make-bsymtype. A symbol with bound argument types is said to *bind* the corresponding arguments in a symtree. That is, if the first argument type of *op* is bound, any occurrence of x in [*op* x ϕ ψ $\ldots$] is bound (here ϕ, ψ, $\ldots$ are arbitrary symtrees). Any occurrence not bound is called *free*.

The only bound symbol type in the Bourbaki Library is 'sv' (for 'set variable'). The Universal Quantifier $\forall$ declared in Section 2 binds its first argument.

Having free variables on the rhs would lead to contradiction. Suppose for example

```
(def "df-foo" sv "foo" () (sv "x")
     [x])
```

Then, using [df-foo x], [df-foo y] and transitivity of set equality, one could prove [= x y] for any set variables x, y. Axioms for primitive symbols with bound arguments must be checked carefully. For example,

```
(prim sv "bar" sv)

(ax "ax-bar" (sv "x")
    (ass [= bar x x]))

(def "df-foo" sv "foo" () (sv "x")
     [bar x])
```

leads to same problem even though x is now bound on the rhs. Also, any subtype of a bound symbol type should be bound.

(Question: are there any easily checkable and sufficiently general conditions for axioms with bound symbols that can be used with sound definitions?).

5.2 Sample Definitions

```
;; The logical disjunction
(def "df-or" pr "\/" (pr ("phi" "psi")) ()
     [-> not phi psi])

;; Existential Quantifier
(def "df-ex" pr "Ex" (sv "x" pr "phi") ()
     [not All x not phi])

;; The class of all sets
;; The class abstraction operator [{ x phi] binds x
(def "df-V" cl "V" () (sv "x")
     [{ x = x x])
```

6 Contexts

For larger projects it gets inconvenient to have all theorems and symbols in the same namespace. Bourbaki provides a hierarchy of namespaces, called *contexts*, to place related symbols and theorems in. At the top of the hierarchy is a single *root context*, the value of *root-context*.

A context is defined with the macro 'in-context'. Example:

```
(in-context "logic"

  (prim pr "->" pr pr)

  ;; Modus ponens
  (ax "ax-mp" (pr ("phi" "psi"))
      (hypo [phi] [!!logic!-> phi psi])
      (ass [psi])))

(in-context "set-theory"

  (ax "ax-infinity" ...)

  ...)
```

The symbol → is defined in context !!*logic* and can be referred to as "!!logic!→" from everywhere. Here the double '!' is an absolute reference, based on the root context. Inside context "logic" !→ or → is enough (one '!' for relative reference; a single '!' in the beginning of a reference can be omitted inside a [ ] -format symtree).

The '!' syntax works outside symtrees also, !!set-theory!ax-infinity returns the Lisp object that can be passed to 'verify' or 'print-theorem'.

Like '[', '!' is a reader macro that is converted to a call to the function 'seek-sym'.

6.1 Nested Contexts

Contexts can be nested up to any level. For example,

```
(in-context "foo"

  (in-context "bar"

    (prim pr "bar-sym"))

  (th "foo-thr" ()
      (ass [!bar!bar-sym ...])
      ...))

(th "other-thr" ()
    (ass [!foo!bar!bar-sym ...]) ; or [!!for!bar!...]
    ...)
```

In fact, corresponding to each theorem (and axiom) a context with the same name is created (or actually, the theorem *is* also the context). It is thus possible to write

```
(in-context "ctx"

  (th "thm" (pr "phi" pr "psi")

      (th "lemma" ...)
      ...
```

```
    (proof
      [lemma phi psi]
      ...)))
```

and the complete name of ”lemma” is ”!!ctx!thm!lemma”. The ‘in-context’ can be used several times with the same context, probably in different files. The context is created if it did not already exist. (It is also possible, though not recommended, to ‘re-open’ a theorem context with in-context:

```
(th "foo" (pr ("phi" "psi")) ...)

(in-context "foo"
  ;; add a hypothesis to "foo"
  (hypo [-> phi psi])) ).
```

A theorem's variables (and dummy variables) reside in the theorem context, but can *not* be accessed from outside, ”!!foo!ϕ” in the previous example would not make sense when there is no wff substituted for ϕ. Attempting to make such a reference results in an error message.

6.1.1 Nested Variables

What about trying to use a lemma or a locally defined symbol from outside the containing theorem? In

```
(th "foo" (pr ("phi" "psi"))

    (def "df-mysym" pr "my-sym" () ()
         [-> phi psi])

    (th "lemma" (pr "chi")
        (ass [<-> phi /\ psi chi])))
```

the value of ”my-var” and the assertion of ”lemma” depend on the arguments supplied to ”foo”. Therefore, when used from outside ”foo”, we must supply arguments to ”foo” also:

```
(th "bar" (pr ("x" "y" "z")) ...
    ;; Use lemma from "foo"
    (proof
      [!!foo!lemma x y z] ; inserts [<-> x /\ y z] to proof list

      ;; Logically equivalent to [ax-mp [-> x y] [z]]
      [!!logic!ax-mp [!!foo!my-sym x y] [z]]
      ...))
```

Trying to use simply "!!foo!lemma" would result in a parse error. Note that the hypotheses for "foo" (if any) must also be satisfied if "lemma" is used this way. There can be multiple levels of nested theorems; arguments for each level must be supplied when accessed from outside.

6.2 Theories

The macro 'theory' defines a context with variables and hypotheses, with same access rules as theorems. Instead of writing

```
(in-context "topology"

  (th "th1" (set "X")
      (hypo [topological-space X])
      ....)
  (th "th2" (set "X")
      (hypo [topological-space X])
      ...)
  ...)
```

one can write

```
(theory "topology" (set "X")
   (hypo [topological-space X])

   (th "th1" () ...)
   (th "th2" () ...))
```

to the same effect. In the latter case the symbol "topological-space" must be defined outside context "topology". The syntax of 'theory' is the same as that of 'th' except that an assertion or a proof makes no sense for a theory (again, these are not banned by Bourbaki). The theory context can be re-opened with 'in-context'.

6.3 Importing and Exporting

It gets tedious to write "!!logic!ax-mp" all the time. To solve this problem, Bourbaki allows *importing* symbols and theorems from other contexts. Actually, importing gives more, "default substitutes" can be provided for some of the variables. An example:

```
;; Theory of topological spaces
(theory "topology" (set "X")
  (hypo [topological-space X])

  ;; The closure of A with respect to the topology of X
  (def "df-closure" set "closure" (set "A") ...)
  (th "th1" ())
  ...)

;; Theory of metric spaces
(theory "metric" (set "d")
  (hypo [metric-space X])

  ;; Make all symbols and theorems from "topology" available.
  ;; The name ‘import’ is reserved by Lisp.
  (bimport [!!topology [metric-topology d]])

  (th "metric-th" (set "A") ...
      ;; Same as [... !!topology!closure [metric-topology d] A ...]
      (ass [... closure A ...])
      (proof
        ;; Same as [!!topology!th1 metric-topology d]
        ;; The hypothesis [topological-space metric-topology d]
        ;; must still be (explicitly) satisfied
        [th1])))
```

In the previous example, trying to use "!!metric!closure" won't work: The symbol ‘closure’ is still defined in "topology" and only accessible with a shorthand in "metric". If instead of ‘bimport’, ‘bexport’ were used, the effect would be (almost) the same as if copies of theorems and symbols were written in "metric":

```
(theory "metric" (set "d")
  (bexport [!!topology [metric-topology d]])
  ...)

(th "some-thr" ()
    (ass [... !!metric!closure my-metric ...])
    ...)
```

There is still only one symbol ‘closure’ and the following symtrees are equal (in the sense of ‘wff-equal’):

```
[!!metric!closure my-metric A]
[!!topology!closure [metric-topology my-metric] A]
```

Exporting is transitive:

```
(theory "topology" (set "X")
  (def "df-clos" set "clos" set))

(theory "metric" (set "d")
   (bexport [!!topology [m-topology d]]))

(theory "banach" (set "B")
   (bexport [!!metric [B-metric B]]))

;; Same as [!!topology!clos [m-topology B-metric B] my-B-space]
[!!banach!clos my-B-space]
```

6.4 Loading Files

If an absolute reference (as in ”!!logic!id”) is made to a non-existent context, Bourbaki tries to load a corresponding file (”logic.lisp”) from the directory specified by *bourbaki-library-path* (default value ”lib” relative to the working directory of Bourbaki) before seeking the symbol again. Additional files may be loaded with the function ‘bload’.

In the following example, the file ”prop.lisp” could be loaded in an another context and a different symbol type (perhaps to prove theorems in Boolean algebras with the same form):

```
;;;; logic.lisp

(setq pr (make-bsymtype))

(in-context "logic"
  (bload "prop.lisp") ; Propositional calculus
  (bload "pred.lisp") ; Predicate calculus
  (bload "equ.lisp")) ; Equality
```

7 Metatheorems

We establish theorems wholesale, by arguments which show that the appropriate sequences *could* be found for each particular case. Such principles, describing general circumstances under which statements are theorems, will be called *metatheorems*.

W. V. Quine[2]

A metatheorem is then, effectively, an *algorithm* for generating theorems. In Bourbaki a (Lisp) function returning theorems as values is called a metatheorem.

Bourbaki contains some tools for making writing metatheorems easier.

7.1 Copying Theorems

The macro ‘copy-th’ creates a new theorem context like ‘th’. Unlike ‘th’ it copies the hypotheses, variables and distinct variable conditions from an existing theorem as specified.

Macro ‘var’ is used to add a single new variable to the end of a theorem's variable list.

Function ‘mkvarlist’ is useful when referring to the copied theorem in a proof.

7.2 Pattern matching

Bourbaki ”borrows” a pattern matching facility from Paul Graham's *On Lisp*[3], slightly modified. An expression

```
(if-match ("->" x y) tree
  (do-something x y)
  (do-something-else))
```

checks that *tree* begins with the symbol $\rightarrow$. If this is the case, do-something is called with x and y bound to the corresponding subtrees of *tree*. Otherwise do-something-else is called.

7.3 The metatheorem macro

The macro 'metatheorem' creates a wrapper around a function taking a single theorem as argument and returning another theorem. The resulting theorem can easily be used from proofs. Additionally, the results are memoized so that the same theorem is returned each time when called with the same argument.

A simple example using most of these features follows. Here a new feature of the [ ] syntax is used: forms beginning with '(' or ',' are given to the Lisp reader:

```
;;; Most of Metamath's "-i" theorems can be replaced with this
;;; Takes a theorem of the form h_i => [-> a b]
;;; Returns the theorem h_i, a => b
;;; "i" is appended to the argument theorem's name
(metatheorem infer (thr)
  ;; Make sure "->" refers to the correct symbol
  (in-context !!logic
    ;; Try to match thr's assertion
    (if-match ("->" x y) (context-assert thr)
      ;; Copy hypotheses, variables and distinct
      ;; variable conditions from thr
      (copy-th thr "i" (hypo vars distinct)
        ;; Add a new hypothesis
        (hypo x)
        (ass y)
        (proof
          ;; refer to the original theorem
          `(,thr ,@(mkvarlist (context-vars thr)))
          [ax-mp ,x ,y]))
      ;; Otherwise go to debugger
      (error "infer: incorrect theorem form"))))

;;; A shorter proof of "id" using infer
(th "id" (pr "phi")
    (ass [-> phi phi])
    (proof
      [ax1 [phi] [-> phi phi]]
      [(infer !ax2) [phi] [-> phi phi] [phi]]
      [ax1 [phi] [phi]]
      [ax-mp [-> phi -> phi phi] [-> phi phi]]))
```

7.4 Possibilities

- The (weak) Deduction Theorem
- Equality theorems of the form $[\rightarrow = \ x\ y\ \leftrightarrow\ \phi(x)\ \phi(y)]$ for general ϕ.
- Algebraic identities, integral formulae, ...
- Metatheorems are to Bourbaki what macros are to Lisp

8 The Bourbaki Library

At the moment only a dozen of theorems in "prop.lisp" (propositional calculus) are in version 3.5 format. The files "pred.lisp" (predicate calculus) and "equ.lisp" (equality) await conversion. These are more or less directly copied from set.mm, with some attempt of systematic naming.

In addition, the files "class.lisp" (class abstraction), "descr.lisp" (the ι-descriptor, the empty set, relation between sets and classes), "pair.lisp" (ordered pairs) and "funct.lisp" (relations and functions) contain the beginnings of set theory, based on Bernays' *Axiomatic set theory*[4]. These also need to be converted.

References

[1] Norman Megill: Metamath, p. 95 (available from http://metamath.org)

[2] Willard Van Orman Quine: Mathematical logic, Harvard University Press 2003 (revised edition), p. 89

[3] Paul Graham: On Lisp, Prentice Hall 1993, Chapter 18

[4] Paul Bernays: Axiomatic set theory. North-Holland Publishing Company 1968