Bourbaki Proof Checker

Juha Arpiainen

December 20, 2006

Contents

1 Introduction

Bourbaki is a program for writing and verifying formal mathematical proofs. It is inspired mainly by Norman Megill's *Metamath*[1] and the related programs Ghilbert[2] by Raph Levien and Mmj2[3] by Mel O'Cat, but aims on providing more powerful syntax and tools for automated proof writing.

Nicolas Bourbaki is the pseudonym of a group of mathematicians, who have written a rigorous multi-volume treatise of Mathematics emphasizing the concept of mathematical structures.[1] My long-term goal with Bourbaki is an online hyperlinked, computer-verified encyclopedia of Mathematics, perhaps in conjunction with the Hyperreal Dictionary of Mathematics project.[4]

Bourbaki is implemented as an embedded language on top of ANSI Common Lisp. This means most Lisp features are available for the user; in particular, it is possible to write Lisp functions that construct proofs (See section 9).

The code of Bourbaki is roughly divided in two levels. The *symbol level* code implements a simple proof system. The *reference level* code provides a hierarchical namespace system to organize the mathematical content. These terms refer to the principal data types used on the respective levels.

Sections 2 to 4 of this document describe the symbol level part of Bourbaki. The reference level code is described in sections 5 to 7. More advanced, programmable features are described in sections 8 and 9.

1.1 Quick start

```
Extract the archive:
$ tar xzvf bourbaki.tar.gz
$ cd bourbaki

Start Lisp:
$ clisp

Load Bourbaki:
```

[1] http://www.metamath.org
[2] http://ghilbert.org
[3] http://planetx.cc.vt.edu/AsteroidMeta/mmj2
[4] http://planetx.cc.vt.edu/AsteroidMeta/HDM

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

Load and verify the propositional calculus module:
> (verify !!prop)
```

1.2 Note on definitions

We refer to the Common Lisp specification[2] for definitions of terms such as *form* (anything that can be evaluated by Lisp) and *string*.

New terms will be defined in BNF syntax. For example,

strings := *string* | (*string**)

defines *strings* to be either a string or a list of strings.

2 Symbols and Formulae

Mathematical expressions are treated as trees of symbols in Bourbaki. Thus the expression $2(a+b)$ is stored internally as $(*_s \ \ (2_s)\ (+_s \ \ (a_s)\ (b_s)))$, where the $*_s, 2_s, \ldots$ are Lisp objects corresponding to the multiplication operator, the number 2, etc.

One usually calls the function `symtree-parse` to create such expressions. Calls to `symtree-parse` are abbreviated with the `[ ]` syntax: evaluating the form `[* 2 + a b]` would construct the above tree, assuming that the symbols have been properly declared before. In this document we will sometimes use the `[ ]` notation to denote the resulting symtree, although this is technically incorrect.

Note that we are using the so-called *Polish prefix notation*, where parentheses can be omitted if each operator takes a fixed number of arguments (which is usually the case in Bourbaki).

2.1 Symbol kinds

Each expression in Bourbaki has a *syntactic type* or *kind*. For example, expressions of first order logic are *variables*, *terms* or *well formed formulae* (*wffs*). Usually one type (the wff) is special in that all *propositions* of the system have this type. The definition feature (section 4) implicitly assumes a wff type, otherwise the wff type is not treated specially.

Syntactic types are declared with the macro `symkind`, for example,

```
(symkind "WFF")
```

A syntactic type may be declared a subtype of another type by giving the `:super` keyword to `symkind`. For example, the line

```
(symkind "VAR" :super term)
```

says that any variable is also a term. Note that the symkind names should be in upper case due to the way Lisp treats symbol names.

The function `(subkindp x y)` returns true if `x` and `y` are the same type or if `x` is a subtype of `y`. An expression e can be substituted for a variable v if `(subkindp (`*`type of e`*`) (`*`type of v`*`))` is true. If this is the case we say e is of *correct type* for v.

2.2 Primitive symbols

Primitive symbols are declared with macro `prim`:

```
prim-form := (prim symkind name ( varspec ) body)
name      := string
varspec   := { typespec strings }*
typespec  := symkind | (:bound symkind)
body      := form*
```

For example, the declaration

```
(prim wff "=" (term "x" term "y"))
```

states that `[= x y]` is a wff for any two terms `x` and `y`. Here `x` and `y` are the *variables* of the symbol `=`. The number of variables is called the *arity* of the symbol.

Several variables of the same kind can be declared by putting their names in a list:

```
(prim wff "=" (term ("x" "y")))
```

would have worked as well. Bourbaki defines a reader macro '`![`' for abbreviating lists of strings:

```
(prim wff "=" (term ![x y]))
```

is equivalent to both of the above declarations.

The names of symbols and their variables are case-sensitive. The symbols are stored in a namespace separate from Lisp symbols. A reference to the symbol can be obtained using the '`!`' reader macro: `!=` returns a reference to the equality symbol just defined. Inside brackets the use of '`!`' is optional: `[= a b]` or `[!= !a !b]`.

It is possible to redefine a symbol: the new declaration replaces the original. References to the old symbol are not updated automatically, however. The old and the new symbol are never the same in the sense of `eq` even if they have the same arity and variables of the same types. This feature should be used only while 'debugging' proofs interactively. Identically named symbols would typically cause exportion to other proof systems to fail.

2.3 Symbol trees

A *symbol tree*, or *symtree* for short, is a list (*op* arg_1 arg_2 ... arg_n) such that *op* is a Bourbaki symbol and each arg_i is a symtree. *op* is called the *operator* of the tree and the arg_i are *arguments*. The leaves of the tree are symbols with no arguments ($n = 0$ is allowed in the definition).

Patterns of symtrees are represented with symtree-like objects that have integers or Lisp symbols in place of some of the subtrees, standing for unspecified symtrees. Note that a lone integer or symbol is not a valid symtree pattern.

The *type* of a symtree (*op* arg_1 ... arg_n) is the symbol kind of *op*. The tree is *well formed* if the arity of *op* is n and each arg_i is well formed and of correct type for the corresponding variable of *op*. In this document the term *wff* is used for any well-formed symtree.

Later on we will use "generalized" symtrees that have theorems or other Bourbaki objects as operators (in place of Bourbaki symbols). Most of this section is valid for the generalized symtrees also, and we will call them simply symtrees. The type of a generalized symtree is `nil` if the operator is not a Bourbaki symbol.

2.4 Symtree functions

The well-formedness of a symtree can be tested with the function `wffp`.

Two symtrees can be compared with `wff-equal`. `(wff-equal x y)` returns true if `x` and `y` have isomorphic tree structure and the corresponding symbols are `eq`.

`wff-type` returns the type of a symtree.

A common operation is to simultaneously substitute all occurrences of a number of symbols in a symtree with some other symtrees. This operation is performed by `replace-vars`, a function taking the symtree and a list of pairs (*sym*$_i$. *subst*$_i$) called the *substitution map* or *smap* in the code.

The result of substituting *sym*$_i$ with *subst*$_i$ in the symtree *tr* is denoted by $tr(sym_1/subst_1, \ldots, sym_n/subst_n)$ or $tr(subst_1, \ldots, subst_n)$ if the *sym*$_i$ are clear from context.

`(print-symtree tree stream)` prints the tree using the bracket notation. The default value of `stream` is `*standard-output*`.

3 Theorems and Proofs

Theorems and *axioms* in Bourbaki have zero or more *variables*, *hypotheses* and *assertions*. In addition a theorem must have a *proof*. The variables are Bourbaki symbols of arity zero; the hypotheses and assertions are wffs, usually containing the variables. In the simple case the proof is a sequence of theorem references justifying the assertions (see section 8 for the general case).

The theorem is treated as a transformation rule: when its hypotheses are satisfied for some symtrees substituted for its variables, its (correspondingly substituted) assertions can be used to prove other theorems.

Theorems are defined using the macros `th`, `hypo`, `ass`, and `proof`:

```
theorem-form    := (th name varspec body)
hypothesis-form := (hypo symtree*)
assertion-form  := (ass symtree*)
proof-form      := (proof proof-line*)
proof-line      := symtree
```

Axioms are defined like theorems, but using `ax` instead of `th`:

```
axiom-form := (ax name varspec body)
```

Listing 1 contains an example from propositional calculus.

Listing 1: Beginnings of propositional calculus

```
;; File example1.lisp
(symkind "WFF")
```

```
;; the implication symbol
(prim wff "->" (wff ![x y]))

;; the axioms
(ax "ax1" (wff ![A B])
  (ass [-> A -> B A]))
(ax "ax2" (wff ![A B C])
  (ass [-> -> A -> B C -> -> A B -> A C]))

;; modus ponens
(ax "ax-mp" (wff ![A B])
  (hypo [A] [-> A B])
  (ass [B]))

;; identity law for '->'
;; see below for interpretation of the proof
;; compare with id1 in set.mm
(th "id" (wff "A")
  (ass [-> A A])
  (proof
    [ax1 A [-> A A]]
    [ax2 A [-> A A] A]
    [ax-mp [-> A -> -> A A A]
           [-> -> A -> A A -> A A]]
    [ax1 A A]
    [ax-mp [-> A -> A A] [-> A A]]))
```

As seen from this this example, the `hypo` (and `ass`) lines are optional. Multiple hypothesis and assertions lines can be used:

```
(hypo [A])
(hypo [-> A B])
```

works as well for `ax-mp`. In contrast, each `proof` replaces the previous one. Hypothesis and axiom lines can be freely mixed, and the order usually does not matter. The proof should be after the hypotheses and assertions. Bourbaki allows a `proof` expression for an axiom, although the proof won't be used anywhere.

Theorems and axioms live in the same namespace with symbols. The rules concerning redefinition of symbols apply also to theorems.

A theorem's variables, hypotheses, assertions and proof can be printed with `print-theorem`.

3.1 Verifying proofs

The function `verify` checks the correctness of a theorem or axiom *thr*. The (somewhat simplified) verification algorithm follows:

1. Check that the hypotheses and assertion of *thr* are well-formed.
2. If *thr* is an axiom, we are done. Otherwise initialize list L with the hypotheses of *thr*.
3. For each proof line `[`*ref* $subst_1$ ... $subst_n$`]`,
   - 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 h`(`$var_1/subst_1, \ldots, var_n/subst_n$`)` is in L, where $var_1, \ldots, var_n$ are the variables of *ref*.
   - For each assertion a of *ref*, insert a`(`$subst_1, \ldots, subst_n$`)` into L.
4. Check that each assertion of *thr* is in L.

The verifier detects *vicious circles* (theorems referring to themselves or cycles of theorem references).

3.2 Distinct variables

Sometimes the simple substitution rule of Bourbaki causes problems. Consider for example the theorem $\forall x \exists y (x \neq y)$ of predicate calculus (which is true assuming there are at least two objects). Simple substitution of z for both x and y would result in the incorrect formula $\exists z (z \neq z)$.

In the above theorem x and y should be marked *distinct* using the macro `dist`:

```
(th "exists2" (var ![x y])
  (dist (!x !y))
  (ass [∀ x ∃ y ≠ x y])
  (proof ...))
```

When two variables `x` and `y` are marked distict in a theorem, the substitution of a for `x` and b for `y` is valid only if

- no variable occurs in both a and b, and
- each variable occurring in a is marked distinct from each variable in b.

(in case a and b are more complex expressions the name *disjoint* variable condition might be more appropriate).

Lists of more than two variables can be given to `dist`. All variables in the list are made pairwise distinct. The general form of `dist` is

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

Distinct variable conditions are logically simpler than the concept of a term being 'free for' a variable in a wff.[4] See the Metamath site [5] for details and for an axiomatization of predicate logic using distinct variable conditions.

3.3 Local variables

Sometimes it is necessary to use extra variables in a proof, in addition to those used in hypotheses or assertions. Such 'dummy' or 'local' variables can be defined with macro `loc`, for example `(loc var "z")`.

```
local-form := (loc varspec)
```

Local variables are treated as distinct from each other and all the 'normal' variables.

3.4 Statistics

The function `collect-stats` walks through the proof of its argument theorem *thr* and calculates the following information for *thr* and (recursively) the theorems referenced by *thr*:

- the axioms the theorem depends on

- definitions (see below) used in the proof, either directly or by a referenced theorem
- number of proof steps if the proof were expanded to axioms
- list of theorems directly referencing this theorem.

The function `show-stats` prints the collected information.

Example:

Listing 2: Statistics

```
> (load "doc/example1.lisp")
> (collect-stats !id)
> (show-stats !id)
Depends on axioms: ax-mp ax2 ax1
Uses definitions:
Is used by 0 theorems:
Total proof length: 5
> (show-stats !ax1)
Depends on axioms:
Uses definitions:
Is used by 1 theorem: id
Total proof length: 0
```

4 Definitions

Writing everything in terms of primitive symbols gets tedious quickly. New symbols can be defined in terms of previous ones with the `def` macro.

```
define-form := (def def-name symkind sym-name
                    ( sym-vars ) ( other-vars ) rhs body)
def-name    := string
sym-name    := string
sym-vars    := varspec
other-vars  := varspec
rhs         := symtree
```

This behaves as if a primitive symbol was declared with

```
(prim symkind sym-name ( sym-vars ))
```

together with an axiom

```
(ax def-name (sym-vars other-vars)
  (ass [eq sym-name sym-vars rhs]))
```

The right-hand-side may contain only primitive and already defined symbols, variables of the defined symbol, and *auxiliary variables* (the `other-vars`) that must be *bound* (see section 4.2).

Here `eq` is the *equality operator* for the symbol kind of the defined symbol. For example, the equality operator for wffs is the biconditional '↔'. The equality operator would in this case be specified with

```
(set-symkind-eq-op wff !↔)
```

Trying to define a symbol of a type for which no equality operator is specified is an error.

The equality operator should be of the 'wff' type and have exactly two variables. Note that the equality operator of the wff type (the biconditional) cannot be used to define itself; it must be a primitive symbol in Bourbaki.

4.1 Soundness of definitions

A definition is *sound* if

- each formula containing the defined symbol can be proved (using the definition) to be equivalent to a formula without the symbol, and
- for each theorem proved using the definition, if its hypotheses and assertions do not contain the defined symbol, there is a proof that does not use the definition.

These conditions state that definition does not add anything new to the language: it is just an abbreviation for a pattern of primitive symbols.

Definitions of the form `[eq sym` $var_1 \ldots var_n$ `rhs]` are provably sound if they satisfy certain bound variable conditions (see below) and the equality operator is an *equivalence relation* satisfying the *substitution laws*, that is, the following formulae are theorems:

- `[eq a a]` (Reflexivity)
- `[→ eq a b eq b a]` (Symmetry)
- `[→ ∧ eq a b eq b c eq a c]` (Transitivity)

- `[→ eq a b eq' φ(x/a) φ(x/b)]` for any wff $\phi(x)$. Here *eq'* is the equality operator for the type of ϕ; `φ(x/a)` denotes the formula where `a` is properly substituted for `x` in `φ(x)`.

Ensuring that these conditions hold for *eq* is a responsibility of the user.

4.2 Bound Variables

If the right-hand side of a definition uses auxiliary variables, they must be *bound* in the sense that the value of *rhs* in the formal system does not depend on what is substituted for the auxiliary variables. For example, x is bound in $\exists x(x > y)$ because $\exists x(x > y) \leftrightarrow \exists z(z > y)$ for any variable z distinct from y.

More formally, a symbol s with variables $v_1, \ldots, v_n$ *binds* v_i if the formula

`[eq [s` a_1 ... a_{i-1} v_i a_{i+1} ... a_n`]`
`[s` $a_1(v_i/a_i)$ ... $a_{i-1}(v_i/a_i)$ a_i $a_{i+1}(v_i/a_i)$ ... $a_n(v_i/a_i)$`]]`

is provable for any substitutions $a_1, \ldots, a_n$ of correct type and a_i distinct from the other a_j. In this case we say all occurrences of v_i within `[s` a_1 ... a_{i-1} v_i a_{i+1} ... a_n`]` are *bound by* s.

To declare that a symbol binds some of its variables, one can give an additional `:bound` argument in the *varspec* of that symbol. For example, the universal quantifier should properly be declared as

```
(prim wff "∀" ((:bound var) "x" wff "φ"))
```

The *other-vars* of a definition are implicitly marked `:bound` and distinct from each other and the *sym-vars*.

Like the substitution law in the previous section, Bourbaki cannot generally verify the validity of a `:bound` declaration for a primitive symbol. However, the following check is possible for a defined symbol:

A definition `[eq sym` $var_1 \ldots var_n$ `rhs]` is valid if for every var_i marked `:bound` (and all the auxiliary variables), all the *potential* occurrences of var_i are bound in *rhs*, that is, var_i must be distinct from all variables that occur free in the *rhs*.

This condition is checked by `verify`, otherwise definitions are treated like axioms by the verifier.

4.3 Sample Definitions

Listing 3: Definitions

```
;; Conjunction
(def "df-and" wff "∧" (wff ![φ ψ]) ()
  [¬ → φ ¬ ψ])

;; Existential quantification
(def "df-ex" wff "∃" ((:bound var) "x" wff "φ") ()
  [¬ ∀ x ¬ φ])

;; Proper substitution. Theorem "sb7" in set.mm
;; [sb a x φ] is the wff where a is properly
;; substituted for x in φ.
(def "df-subst" wff "sb" (term "a" var "x" wff "φ") (var "y")
  [∃ y ∧ = y a
      [∃ x ∧ = x y φ]])
```

In the third example, `x` cannot be marked `:bound`. If the term substituted for `a` contains free occurrences of the variable substituted for `x`, those occurrences are free in the `rhs`. If `x` were distinct from `a`, it could be marked `:bound`.

5 The Context System

When writing mathematics, it is usual to omit information that is formally necessary, but "clear from context". For example, when speaking about closed sets in a Hilbert space H, we mean closed with respect to *the topology induced by the metric induced by the norm induced by the inner product of* H.

A *context* in Bourbaki contains among other things bindings for the parser that make it possible to parse `[closed A]` as

```
[closed A [topology-of metric-of norm-of inner-product H]],
```

for example. These bindings are established by *importing* and *exporting* contexts (section 6.4).

In this section we describe the *namespace hierarchy* provided by the contexts: a context may contain symbols, theorems and subcontexts. At the top of the hierarchy there is a *root context*, the value of `*root-context*`. New symbols and theorems declared with `prim`, `th` and so on, are inserted into the *current context*, the value of `*current-context*`. By default,

`*current-context*` is the same as `*root-context*`.

New contexts are created with macros `module` and `local`:

```
module-form := (module name body)
local-form  := (local name body)
name        := string
```

`local` creates a new subcontext of the current context and evaluates the ***body*** expressions in this new context. `module` always creates a *top-level* context, one that is a subcontext of the root contetx.

A sample context hierarchy follows.

Listing 4: Sample context hierarchy

```
;; File example2.lisp
(module "logic"
  (local "propositional"
    (prim wff "→" (wff ![ϕ ψ])))
  (local "predicate")
    (prim wff "∀" ((:bound var) "x" wff "ϕ")))
(module "set-theory"
  (local "zfc"
    (prim wff "∈" (set ![x y]))
    (local "axioms"
      (ax "union" ...)
      (ax "choice" ...))))
```

Symbols and theorems are retrieved from the context hierarchy with the function `seek-sym`, usually abbreviated with the reader macro '`!`'. References beginning with `!/` are *absolute* (with respect to the root context), other references are relative to the current context. In the sample hierarchy, `!/set-theory!zfc!axioms!union` and `!axioms!union` refer to the same axiom if `!/set-theory!zfc` is the current context. The macro `in-context` can be used to bind `*current-context*`:

```
(in-context !/set-theory!zfc!axioms
  (ax "infinity" ...))
```

re-opens the `axioms` context to insert a new axiom.

The symbols and theorems in Bourbaki are actually treated as contexts, and may themselves have subcontexts. For example, lemmas can be conveniently placed as subcontexts of the theorem they are used for:

```
(th "sample" (wff "X")
  (th "lemma1" ...)
  (th "lemma2" ...)
  ...)
```

The words *symbol* and *context* are used somewhat interchangeably in the rest of this document.

The *full name* of a context is the list of names of the ancestors of that context, for example, `("union""axioms""zfc""set-theory")` is the full name of `!/set-theory!zfc!axioms!union`.

5.1 Meta information

Each context contains a table of meta information indexed by Lisp symbols. Some fields are to be filled by the user, others by Bourbaki or other programs. Meta information can be accessed with `(gethash` *key* `(context-meta` *context*`))` The macro `meta` sets meta fields of the current context:

meta-form := `(meta {`*key content*`}*)`
key := *symbol*
content := *form*

Table 1 lists the fields reserved by the current version of Bourbaki. They are all symbols in the `keyword` package.

Key	Set by	Description (reference)
`:bound`	Symbol defining macros	the :bound status of a variable
`:comment`	user	short description
`:def`	`def`	definition corresponding to a symbol
`:defs-used`	`collect-stats`	
`:depends-on`	`collect-stats`	axioms this theorem's proof depends on
`:description`	user	longer textual description
`:full-name`	`create-context`	the full name of this context
`:parent`	`create-context`	parent context in the hierarchy
`:proof-length`	`collect-stats`	length of proof when expanded to axioms
`:used-by`	`collect-stats`	list of theorems using this one in a proof

Table 1: Meta keys used by Bourbaki

5.2 Nested variables and hypotheses

In mathematical theories it is common to have theorems with common variables and hypotheses. For example, theorems in the theory of topological spaces would all have a variable `X` and a hypothesis `[topo-space X]`. Common variables and hypotheses can be moved to the parent context:

Listing 5: Topological spaces

```
(def wff "topo-space" (set "X") ...)

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

  ;; closure of A in the topology of X
  (def set "closure" (set "A") ...)

  ;; Theorem: closure is idempotent
  (th "clos-idem" (set "A")
    (ass [= [closure closure A] [closure A]])
    (proof ...)))
```

The macro `theory` is just a version of `local` taking a *`varspec`*. When used from outside of `topology` this behaves as the following were written instead:

```
(local "topology"
  (def set "closure" (set "X" set "A") ...)
  (th "clos-idem" (set "X" set "Y")
    (hypo [topo-space X])
    ...))
```

Within `topology` the space `X` is used by default when referring to other theorems and definitions in `topology`

```
> (in-context !/topology
    (print-symtree [closure ∅]))
[closure X ∅]
```

This is a case of *importing* symbols, as defined in section 6.4. When variables from multiple levels are nested, the variables of the innermost context are the *rightmost* ones.

Note that definitions, unlike axioms and theorems, do *not* inherit hypotheses from the parent context. This is to ensure the definition can always be eliminated.

5.3 Context functions

New contexts are created with function `create-context`:

```
(create-context :name string :parent context
                :type symkind :class symbol)
```

`parent` defaults to `*current-context*`; a `nil` parent creates a context outside the hierarchy. `class` should be one of the symbols `:axiom`, `:definition`, `:theorem`, `:prim`, `:sym`, `:arg` (a variable), `:loc` (a dummy variable), `:context`, or `:module`.

`create-context` does not yet insert the new context to `parent`'s symbol table, see `insert-sym` in section 6.

Usually `create-context` is not called directly, but context creating macros such as `th` and `prim` are used instead. `defcontext` is the general form of these macros:

```
defcontext-form := (defcontext class name ( varspec ) body)
```

Like `create-context`, `defcontext` does not insert to parent's symbol table.

`mkrootcontext` creates an empty context without a parent, a suitable value for `*root-context*`. `flush` sets `*root-context*` to a fresh root context, usually sending the previous hierarchy to the garbage collector.

All symbols and theorems in Bourbaki are structures of type `context`:

```
(defstruct context
  name
  type
  class
  (meta (make-hash-table 'eq))

  vars
  hypo
  assert
  distinct
  proof
```

```
  (syms (make-hash-table 'equal))
  imports
  exports)
```

The `syms`, `imports` and `exports` are described in section 6.

There is some redundancy. The `proof`, for example, does not make sense for a primitive symbol. Future versions of Bourbaki might take a more object-orinted approach.

6 Symbol References

A *symbol reference* or *symref* is a function taking symtrees as arguments and returning a symtree. Symrefs are used to represent argument transformations. For example, in the hierarchy of listing 5 above, the call

```
(in-context !/topology
  !closure)
```

returns a symref taking one argument. When applied to a symtree `A` this symref returns `[closure X A]`.

6.1 Terminology

The symbol references are structures of type `symref`:

```
(defstruct symref
  target
  arity
  fn
  proof)
```

where `target` is the referred context (or `nil`, if the symref does not refer to a specific context), and `fn` is a Lisp function used to construct arguments for the target context (`fn` should return a list of symtrees with length equal to `target`'s arity). `arity` is an integer n describing the function `fn`.

If $n \geq 0$, `fn` is a function taking some *fixed number* $m \geq n$ of arguments. The symref is *exact* if $m = n$.

If $n < 0$, `fn` is a function taking a *variable number* of arguments, at least $|n| - 1$.

A symref is *composable* if `arity` ≥ 0 and the `target` is not `nil`. In the rest of this section we assume all symrefs to be composable. Symrefs with variable number of arguments and `nil` target appear in section 9.

The function `(mkcontext x)` returns `x` itself if it is a context, and the target of `x` if `x` is a composable symref. Many of the Bourbaki functions taking context arguments also accept composable symrefs, calling `mkcontext` if necessary. `verify` and `print-theorem` are examples of these.

Given an exact symref r with `arity =` n and some symtrees $t_1, \ldots, t_n$ we can construct the symtree with r's target as operator and the results of `(funcall (symref-fn` r`)` $t_1 \ldots t_n$`)` as arguments. The function `apply-ref` performs this operation:

```
(defun apply-ref (ref rest args)
  (cons (symref-target ref)
        (apply (symref-fn ref) args)))
```

When speaking about *applying* a symref, we mean this operation.

6.2 Composing References

The fundamental operation of symrefs is composition. Given a composable symref `f` of arity n and a symref `g` of arity m, `(compose-ref f g)` has arity $n + m$. It is exact if and only if `f` is, and composable if and only if `g` is. The target of `(compose-ref f g)` is the target of `g`.

When the function of `(compose-ref f g)` is called with arguments $a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m$, the first n arguments are given to `f`. `f` should return a list $c_1, c_2, \ldots, c_k$ of symtrees, where $m + k$ is the actual number of arguments g expects. The result of calling `(compose-ref f g)` is then the result of calling `g` with $c_1, \ldots, c_k, b_1, \ldots, b_m$.

Corresponding to a theorem

```
(th "example" (type_1 "var_1" ... type_N "var_N")
    ...)
```

the `syms` hashtable of the parent context contains a symref with `arity = N` under the key `"example"`. The symref function in this case just returns its arguments unmodified. Contexts of other types are inserted into the symbol table in the same way. Note that if an ancestor context of `"example"` has variables, the arity of `"example"` is greater than N and the symref is not exact

(the symref function should be called with all arguments for the target, not just the innermost ones).

A reference `!/x!y!z!example` composes these subcontext references while going down the context hierarchy, resulting in an exact symref with `"example"`'s arity.

The composition of symrefs is an associative operation as long as all te compositions are legal.

6.3 The Parser

The function `symtree-parse` takes arguments of various types and attempts to construct a symtree from them. If the first argument *op* to `symtree-parse` is a symref with `arity` $= n \geq 0$, *op* is applied to the next n subtrees parsed recursively (*op* must be an exact symref). If $n < 0$, subtrees are parsed until all the arguments are used up. *op* is then applied to all of the parsed subexpressions.

If the first argument is of any other type, it is immediately returned. This allows construction of symtree patterns that have integers or Lisp symbols in place of subtrees.

6.3.1 Examples

`(symtree-parse !* !+ !a !b !- !a !b)` finds that `!*` is a symref requiring two arguments. It then parses the subtrees `[+ a b]` and `[- a b]` and finally applies `!*` to them, resulting in `[* [+ a b] [- a b]]`.

```
(symtree-parse !*
               (symtree-parse !+ !a !b)
               (symtree-parse !- !a !b))
```

returns the same symtree as the first example: when getting arguments for `!*`, `symtree-parse` sees the two lists from the nested calls and passes them to `!*`.

`(symtree-parse !* 0 1)` returns the symtree pattern $(*_s$ `0 1`$)$.

6.3.2 Bracket syntax

The reader macro '`[`' is an abbreviation for `symtree-parse`. Whitespace-separated tokens are read until a '`]`' is seen. Tokens beginning with one of the characters '`[`', '`!`', '`"`', '`(`', or '`,`' receive a special treatment. Otherwise

the token is a string naming a symbol in `*current-context*`. Thus `[+ x y]` is an abbreviation for

```
(symtree-parse !+ !x !y)
```

The special characters are treated as follows:

- `[` begins a nested symtree. Using this construct is necessary to disambiguate expressions with operators taking a variable number of arguments. It may be used at any time to parenthesize the expression for clarity.
- `!` begins a call to `seek-sym` read with the usual syntax of '`!`'.
- `"` reads a symbol name using Lisp syntax for strings. It may be used to escape the special characters in a symbol's name, for example, `[+ "[foo␣bar]" 3]`.
- When a '`,`' is encountered, the Lisp reader is used to read a form. The form is then evaluated and given to `symtree-parse`. This works like with backquote. Example:

  ```
  (let ((x [+ a b]) (y [- a b]))
    [* ,x ,y])
  ```

  produces the same symtree as `[* + a b - a b]`. `[-> ,1 ,2]` constructs a symtree pattern with integers.

6.4 Importing and Exporting

We now return to the example of topology in Hilbert spaces mentioned in section 5, using topological closure as an example. Begin from the theories of topological and metric spaces:

```
(theory "Topo" (set "X")
  (hypo [topo-space X])

  ;; Define closure of A in the topology of X
  (def "df-clos" set "clos" (set "A") ...)

  ;; Sample theorem: closure is idempotent
  (th "clos-idem" (set "A")
```

```
    (ass [= clos clos A clos A])))

(theory "Metric" (set "d")
  (hypo [metric d])

  ;; Define the topology induced by d
  (def "df-metric-topo" set "topology-of" () ...))
```

Now, to make use of the topology theorems for metric spaces, *import* the context Topo:

```
(in-context !/Metric
  (import [Topo topology-of]))
```

This creates an exact symref i of arity 0 with the function

```
(lambda () (list [topology-of d]))
```

Now, when seek-sym tries to find a symbol named "clos" it looks not only at the current context `!/Metric`, but also the imported context `!/Topo`. The 1-argument symref `!clos` is composed with i resulting in an exact symref taking an argument `A` and returning the list `([topology-of d] A)`.

Imported symbols are only visible from the importing context. It is not possible to use `!/Metric!clos` outside the context `!/Metric`. To achieve this, we should use `export` instead of `import`:

```
(in-context !/Metric
  (export [Topo topology-of]))
```

Now `!/Metric!clos` can be used from everywhere as the closure operator specialized to metric spaces: `[!/Metric!clos d A]` is the closure of `A` in the topology induced by the metric `d`. The statement `(export [Topo topology-of])` creates a symref e with arity 0 and function

```
(lambda (x) [topology-of ,x])
```

(Note that e is not an exact symref.) When parsing `[!/Metric!clos d A]`, the symref `!Metric` is composed with e and `!clos` yielding finally an exact symref with arity 2 and function

```
(lambda (x y) [!Topo!closure [topology-of ,x] ,y])
```

The intended use of `export` is to provide specialized or extended versions of theories, while `import` is used to abbreviate references within a context.

For example, most contexts will import propositional and predicate calculus for easy use of the logic symbols.

We can now define the theories of normed and Hilbert spaces in the same way:

```
(theory "Normed" (set "N")
  (hypo [Normed-space N])

  (def "df-metric" set "metric-of" () ...)
  (export [Metric metric-of]))

(theory "Hilbert" (set "H")
  (hypo [H-space H])

  (def "df-norm" set "norm-of" () ...)
  (export [Normed norm-of]))
```

Here the statement `(export [Metric ...])` exports not only `Metric`, but also all contexts exported from `Metric` by composing the symrefs associated with the `export` statements. Thus `Normed` will export all symbols from `Topo` with the symref

```
(lambda (x) [topology-of metric-of ,x])
```

Likewise, `Hilbert` will export symbols from `Topo`, `Metric` and `Normed`.

The Bourbaki `import` and `export` are analoguous to the Lisp functions with the same names; both deal with visibility of symbols. There are differences, however: Bourbaki import and export deal with whole contexts instead of individual symbols. The symbols and theorems defined within a context are also exported by default.

More generally, default values can be given to just some of the variables of the imported context by using `import` or `export` with a symtree pattern. Take the theory of continuous functions, for example:

```
(theory "Cont" (set ![X Y])
  ;; f: X -> Y is continuous
  (def "df-cont" wff "continuous" (set "f") ...)

  ...)
```

We can specialize this to continuous real-valued functions with

```
(local "real-valued"
  (import [Cont ,0 ℝ]))

(in-context !real-valued
  (print-symtree [continuous A f]))
⇒ [continuous A ℝ f]
```

and to paths (continuous functions from the unit interval I) with `(import [Cont I ,0])`.

6.5 Importing and exporting with proofs

In the account of previous section we have skipped over one detail: `Topo` has hypothesis `[topo-space X]` while `Hilbert` has `[H-space H]`. It would be nice if the proof fragment corresponding to Hilbert ⇒ Normed ⇒ Metric ⇒ Topology could be stored with the exports and inserted automatically whenever a topology theorem is used in a Hilbert space proof.

The optional `:proof` argument of `import` and `export` does just this, as illustrated in the following listing (see section 8 for `linear-subproof`):

```
(theory "Metric" (set "d")
  (hypo [metric d])

  ;; Define the topology induced by d
  (def "df-metric-topo" set "topology-of" () ...)

  ;; A conversion theorem:
  ;; the induced topology is a topological space
  (th "Metric-to-Topo" ()
    (ass [topo-space topology-of])
    (proof ...))

  ;; Export !/Topo with a proof
  (export [!/Topo topology-of]
          :proof (linear-subproof ([metric d])
                                  ([topo-space topology-of])
                    [Metric-to-Topo])))
```

When multiple symrefs with proofs are composed, the resulting proof will be a concatenation of the original proof fragments. The proof is used whenever a reference to an axiom, theorem or definition is made via the symref

within the proof of a theorem (or whenever the variable `*current-proof*` is set).

6.6 Seeking symbols

The function `seek-sym` seeks a symbol by name from the context hierarchy, returning a symref. The first argument to `seek-sym` is either `:rel` or `:abs` specifying search from the current context or the root.

In a call `(seek-sym :rel "name1""name2"... "nameN")` the first name is searched from the *imports* of `*current-context*`. If a symbol s_1 is found via an import i, `"name2"` is searched from the *exports* of s_1. For each further name `seek-sym` takes one step "sideways"through an export and one step "down" to find the next name. `seek-sym` composes the symrefs it follows returning finally a symref with target a symbol named `nameN`. In an absolute reference the first name is searched from the imports of `*root-context*` instead.

The *exports* of a context K are searched in the following order:

1. K always exports itself with an identity symref
2. If K exports other contexts $C_1, \ldots, C_n$ in this order, seek from the export lists of the C_i in *reverse order* (first C_n, then contexts exported by C_n, then C_{n-1}, and so on).

The *imports* of a context K are searched in the following order:

1. K imports itself with a constant function returning the list of variables of K.
2. If K imports contexts $C_1, \ldots, C_n$, seek from the *export* lists of the C_i in *reverse order*.
3. Seek from the import list of the parent of K.

Note that the list of exports is always a subset of the list of imports. A statement `(export [X])` adds `X` to both lists.

The reader macro '`!`' abbreviates calls to `seek-sym`. After the initial '`!`' or '`!/`' exclamation mark separated tokens are read until whitespace is seen. Special characters in a token can be escaped by enclosing it in double quotes.

6.7 Aliases

The imports and exports just described provide argument transformations for whole contexts. Sometimes it is useful to abbreviate just a single symbol or pattern of symbols in this way.

For example, binary relations are encoded as sets of ordered pairs in set theory. Thus we would have to write $\langle x, y\rangle \in \leq_{set}$ or `[∈ pair x y` $\leq_{set}$`]` for the usual $x \leq y$.

We could of course introduce a definition

```
(def "df-less" wff "<=" (set ![x y]) ()
  [∈ [pair x y] ≤])
```

but this has the inconvenience that '`<=`' is a new primitive symbol from the point of view of the verifier. Specialized versions of the general properties of binary relations would have to be proved anew for every symbol introduced in this way.

Instead we can introduce an *alias* to the symbol table:

```
(alias "<=" (set ![x y])
  [∈ [pair x y] ≤])
```

The name '`<=`' is now associated with a symref `(lambda (x y) [∈ [pair ,x ,y] ≤])`. The verifier never sees a symbol named '`<=`'.

The macro `pattern` constructs and returns this symref without inserting it into the symbol table:

```
(pattern (set ![x y])
  [∈ [pair x y] ≤])
```

7 Modules and Files

It is common to have theorems of similar form, but different symbols in their assertions. Taking the partial orders (actually lattices) $\leq$ and $\subseteq$ as an example, we have theorem pairs such as '$x \leq y \Leftrightarrow x = \min(x, y)$' and '$x \subseteq y \Leftrightarrow x = x \cap y$'. The following listing exhibits the characteristic properties of these relations using a generic "object" symkind.

Listing 6: Partial orders

```
;;; File order.lisp
```

```
(module "order"
  (symkind "OBJ")

  (prim wff "≤" (obj ![a b]))

  ;; Define the associated strict order
  (def "df-less" wff "<" (obj ![a b])
    [∧ ≤ a b ≠ a b])

  (def "df-meet" obj "/\\" (obj ![a b]) ...)
  (def "df-join" obj "\/" (obj ![a b]) ...)

  ;;; Axioms for partial orders
  ;; Antisymmetric law
  (ax "antisym" (obj ![a b])
    (ass [→ ∧ ≤ a b ≤ b a = a b]))

  ;; Transitive law
  (ax "tr" (obj ![a b c])
    (ass [→ ∧ ≤ a b ≤ b c ≤ a c]))

  ;;; Theorems for partial orders
  (th "meet-le" (obj ![a b])
    (ass [↔ [≤ a b] [= a /\ a b]]))

  ...)
```

We would now like to *interpret* `!/order` in the language of set theory to get versions of the theorems with $x \subseteq y$ and $\langle x, y\rangle \in \leq$ where `!/order` has `[≤ x y]`.

The solution is to load `order.lisp` in such a way that Bourbaki sees *aliases* for the relevant symbols instead of the declarations in `order.lisp`. We now describe the functions `require` and `providing` that are used to set up this interpretation.

7.1 Require

The function (require *name module*) looks for a top-level context *name*. If such a context exists, a reference to it is returned as if !/*name* was used. Otherwise the file *module*.lisp is loaded (from the directory *bourbaki-db-path* whose default value is the "lib" subdirectory). The file should define a top-level context with the same name *module*.

Before loading the file, require sets *root-context* to a newly created context. This new root context is made to import the previous one. *module*.lisp is thus free to define any additional top-level contexts without cluttering the namespace of the module requiring it. This is similar to Java, where a .java file can contain one public class with the same name, and additionally any number of private classes.

After the file is loaded, require inserts the symref to *module* to the previous root context's symbol table under the key *name* and also returns this symref.

Usually require is called with both arguments the same. The reader macro !!*module* is an abbreviation for (require *module module*). This leads to the idiomatic expression (import [!!*module*]) where the module is loaded and imported at the same time.

A case where the *name* argument is needed would be second order logic, where we have predicate calculus for two kinds of variables ranging over objects and collections. There we would use something like

```
(require "obj" "predicate")
(require "coll" "predicate")
```

Returning to the example of partial orderds, we now move the axioms and definitions to their own file:

```
;;; File order-ax.lisp

(module "order-ax"
  (symkind "OBJ")

  (prim wff "≤" (obj ![a b]))

  ;; other primitive symbols, axioms and definitions as before
  )
```

order.lisp will then require the axiom file:

```
;;; File order.lisp

(module "order"
  (symkind "OBJ")
  (export [!!order-ax])

  ;;; Theorems for partial orders
  (th "meet-le" (obj ![a b])
    (ass [↔ [≤ a b] [= a /\ a b]]))

  ;; other theorems as before

  )
```

Note that symbol kinds are stored in the symbol table of each root context. Both files need to declare `(symkind "OBJ")` to ensure they use compatible types. The declaration in `order-ax.lisp` sees that a symkind with the same name is already visible (through the import of `order.lisp`'s root context) and does not try to redefine the type. If the declaration was omitted, `order-ax.lisp` could not be verified standalone. If the declaration in `order.lisp` was omitted, the `OBJ` from `order-ax.lisp` would *not* be copied into the root context of `order.lisp` and the theorems would not parse.

The only thing remaining to interpret the theory of partial orders is to set `order-ax` and `OBJ` properly before using `(require "order")`.

7.2 Providing

The macro `providing` evaluates its body within a temporary root context that has a specified set of top-level bindings:

provide-form := (providing (*binding**) *body*)
binding := (*name symref*)

Now the set theory file can make use of `order.lisp`:

```
;;; file set.lisp

(module "set"
  (symkind "SET")
```

```
;; ...

;; properties of the subset relation
(local "sub"

  (th "subset-antisym" (set ![A B])
      (ass [→ ∧ ⊆ A B ⊆ B A = A B])
      (proof ...))

  ;; ...

  (let ((prov (defcontext :context "provide-order-ax" ()
                (alias "≤" (set ![a b])
                  [⊆ a b])
                (alias "antisym" (set ![a b])
                  [subset-antisym a b])
                ;; other symbols and axioms in the same way
                )))
    (providing (("order-ax" prov)
                ("OBJ" !/SET))
      (export [!!order])))))
```

Theorems for partial orders specialized to the subset relation are now available as `!/set!sub!meet-le` and so on. Similarly a calculus filed could `require order` with aliases `(alias "≤"(set ![x y]) [∈ pair x y ≤])`. Bourbaki does not currently check that the provided aliases match with the symbols in `order-ax.lisp` (that the alias '≤' points to a symbol with arity two and so on). The parser and the verifier must be relied on to catch any mismatches.

Note that the temporary root context created by `providing` does not import the previous `*root-context*` and the entries in its symbol table are not copied to the root context of `set.lisp`. After the `providing` form finishes, the only way to access the `order` theorems is through the `export` in `!/set!sub`.

In summary, the `require` function is used in two ways:

- Within a `providing` form to interpret an existing theory
- At top level to state what axioms or parameters are needed and what

are their default values

7.3 Writing re-usable modules

Making a module interpretable in many different contexts puts some constraints on it.

- The module should not depend on the exact properties of the symbols in the required modules. The '$\leq$' in `order-ax.lisp` might be an alias expanding to a complex expression rather than being a primitive symbol. This implies that symtrees should not be constructed manually, `symtree-parse` must always be used.
- The module should not try to redefine or add symbols to the required contexts.
- Top-level contexts other than those explicitly required may be visible when the file is being loaded. New "private" modules may always be defined without fearing name clashes, however.
- The module could be loaded multiple times within the same session. If the file defines any global functions or variables, these will be redefined each time the file is loaded. Global references to the Bourbaki context system should not be put outside the system.

8 Structured Proofs

The stricktly linear proofs used so far fails to separate the trivial manipulations necessary in a formal proof from the "interesting" proof steps. Bourbaki actually uses a structured proof format like the one described in [6].

A *structured proof* consists of hypotheses, assertions and *justification*. The justification is either a theorem reference (that is, a symtree whose operator is a theorem) or a sequence of (structured) subproofs. We call these *type 1* and *type 2* proofs, respectively. The proof of a theorem has the same hypotheses and assertions as the theorem. Structured proofs are objects of type `subproof`:

```
(defstruct subproof
  hypo
```

```
assert
ref ; for type 1 proofs
sub ; for type 2 proofs
)
```

8.1 Verifying structured proofs

To check the validity of a structured proof, we maintain lists of wffs proved at each level of the proof. The first list is initialized with the hypotheses of the theorem being proved.

The algorithm for a type 2 proof P goes as follows:

1. create a new empty list L.
2. for each subproof S,
   - check the validity of S
   - add the assertions of S to L.
3. check that each assertion of P is in L or one of the upper level lists
4. discard the list L. Following steps may use only the stated assertions of P, not any intermediate results.

For a type 1 proof P,

1. check that the substituted expressions are well-formed and satisfy the distinct variable conditions of the referred theorem
2. check that the (substituted) hypotheses of the theorem are found in the upper-level lists of proven expressions
3. check that the assertions of P are among the substituted assertions of the theorem.

Note that the hypotheses of a subproof are ignored; they are only hints for humans and programs manipulating the proof.

Any structured proof can be "flattened" and the result will be valid according to the algorithm of section 3.1. The converse is not true: if a structured proof step depends on some intermediate results of a type 2 subproof, the flattened version will be valid even though the original proof is not.

8.2 Parsing proofs

There are several ways to create proof objects in Bourbaki.

The function `parse-subproof` handles several kinds of arguments:

- A theorem reference is wrapped in a type 1 subproof with the obvious hypotheses and assertions.
- An argument of the `subproof` type is returned without modification
- When given an ordinary wff, `parse-subproof` seeks for a theorem proving this wff (see section 8.3).
- When given a symref pointing to a theorem, `parse-subproof` unifies hypotheses of the theorem against the previously proved wffs as in a Metamath-style RPN proof. (Not implemented yet; this requires maintaining the proof stack separately.)

When `symtree-parse` is used within a structured proof to construct a reference to an imported theorem, and the import has an associated proof fragment (section 6.5), a type 2 subproof consisting of the fragment and the intended theorem reference is returned instead.

The most general macro to create type 2 proofs is `subproof`:

subproof-form := (subproof (:hypo *form* :assert *form*) *body*)

The `:hypo` and `:assert` forms should be lists of wffs. Within the *body*, the macros `hypo`, `ass` and `line` are locally defined to add hypotheses, assertions and subproofs to the proof being created. `line` calls `parse-subproof` on its argument.

Within the body of a `subproof` form the variable `*current-proof*` points to the list of assertions proved at earlier steps (including upper levels of the proof). Membership on this list can be tested with `(provenp` *wff*`)`.

The case where the body of a `subproof` form is just a sequence of `line` forms is abbreviated with the macro `linear-subproof`:

linear-form := (linear-subproof (*hypo**) (*assertion**) *line**)
hypo := *symtree*
assertion := *symtree*
line := *theorem-reference* | *symtree* | *subproof*

Each *line* is parsed by `parse-subproof`. The macro `proof` is a version of `linear-subproof` that uses the hypotheses and assertions of `*current-context*` by default and sets the resulting proof object into the `proof` slot of `*current-context*`.

8.3 Seeking theorems

The function `parse-subproof` accepts not only theorem references. When given a wff, it tries to find a theorem proving that wff. For this to be practical references to theorems are stored to a search tree indexed by symbols in the assertions of the theorem. This means that a wff `[-> [\/ x y] [/\ u v]]` needs only be tested against theorems of the forms `[-> a b]` (with `[\/ x y]` substituted for `a` and `[/\ u v]` for `b`), `[-> [\/ a b] c]`, and `[-> [\/ a b] [/\ c d]]`. Each candidate theorem is tested with the same method that `verify` uses so that hypotheses and distinct variable conditions are correctly checked.

Seeking currently works only for theorems whose assertion contains all the variables of the theorem. This excludes inference rules such as modus ponens. To apply modus ponens when proving a wff ϕ we would have to find a wff of the form `[`$\rightarrow$ ψ ϕ`]` among the already proved wffs such that ψ has also been proved. The complexity for finding substitutions for these "extra" variables explodes combinatorically as the number of variables and hypotheses increases.

A theorem must be registered with the function `th-pattern` to make `parse-proof` consider it for seeking.

8.4 Pattern trees

TODO: insert here a description of the trees of wff patterns indexed by symbols; these are used by `parse-subproof` for seeking theorems and `provenp` for testing if a wff has been proved. Several functions in `pattern.lisp` for adding wffs to trees and checking membership. Note that removing wffs from pattern trees is not yet possible.

9 Metasymbols

What we have so far done with symrefs amounts to substitution of variables in symtrees. Why not put more general Lisp functions in the `fn` slot of a symref? For example, we could want a summation operator Σ taking variable number of arguments so that

- `[`Σ `x]` is parsed to `[x]`,
- `[`Σ `a b ... n]` is parsed to `[+ a [+ b [+ ... n]]]`,

- `[Σ]` is parsed to `[0]`.

As a Lisp function we would write

```
(defun sum (&rest terms)
  (case (length terms)
    (0 [0])
    (1 (car terms))
    (otherwise [+ ,(car terms) ,(apply #'sum (cdr terms))])))
```

The macro `metath` can be used to place our function in the context system:

```
(metath "Σ" (&rest terms)
  (labels ((sum (terms)
             (case (length terms)
               (0 [0])
               (1 (car terms))
               (otherwise [+ ,(car terms)
                             ,(funcall #'sum (cdr terms))]))))
    (funcall #'sum terms)))
```

and `[Σ x y z ...]` now works as advertised. The name `metath` refers of course to *metatheorems*, but as in the current case, it can be used to define *metasymbols* of any type.

The call `(metath` *`name`* `(`*`vars`*`)` *`body`*`)` places a symref with the function `(lambda (`*`vars`*`)` *`body`*`)` into the symbol table of the current context under the key *`name`*. The symref has `nil target` and arity -1. When `symtree-parse` encounters a symref with negative arity, it parses subexpressions until it runs out of arguments. The symref function is then applied to all the subexpressions. Unlike ordinary symrefs, the function must return a complete symtree rather than the list of arguments for the `target`. Because of these differences, symrefs created with `metath` cannot be used as the first argument to `compose-ref`.

9.1 Resolving name clashes

Our 'Σ' has a potential problem: the values of `[0]` and `[+ a b]` depend on the symbols visible from `*current-context*` at the time the `[Σ ...]` expression is parsed. The results may be unexpected should the user redefine

'+'. (Sometimes having 'Σ' automatically use the currently visible '+' is just the right thing, however).

When the metasymbol function is called, the variable `*meta-context*` is bound to the value of `*current-context*` at the time the metasymbol was being defined. An unambiguous version of 'Σ' would be

```
(metath "Σ" (&rest terms)
  (in-context *meta-context*
    (labels ((sum (&rest terms)
               ;; the rest as before
      )))))
```

9.2 Working with proofs

Metasymbols constructing proofs or theorems are called *metatheorems*. The following listing illustrates some of the tools Bourbaki provides for writing metatheorems. "`infer`" converts a proof of $\phi \to \psi$ or $\phi \leftrightarrow \psi$ to a proof of ψ under the hypothesis ϕ (or possibly vice versa in the case of a biconditional).

```
(metath "infer" (line)
  ;; treat argument as a subproof
  (let ((prf (parse-subproof line)))
    (in-context *meta-context*
      ;; This works only for proofs with one assertion
      (match-case (car (subproof-assert prf))
        ([-> ,'?x ,'?y]
          (linear-subproof (?x) (?y)
            prf
            [ax-mp ,?x ,?y]))
        ([<-> ,'?x ,'?y]
           (if (provenp ?x)
             (linear-subproof (?x) (?y)
               prf
               ;; convert <-> to ->
               [bi> ,?x ,?y]
               [ax-mp ,?x ,?y])
             (if (provenp ?y)
               (linear-subproof (?y) (?x)
                 prf
```

```
;; convert <-> to <-
[bi< ,?x ,?y]
[ax-mp ,?y ,?x])))))))))
```

This uses the pattern matching version of `case`, a similar macro `if-match` is also defined. Note that in the case of a biconditional the metatheorem tests which direction is intended by using the list `*current-proof*` of proven assertions.

The `"infer"` metatheorems saves the user from invoking modus ponens explicitly. It would be used like `[infer ax2 X Y Z]`. Compare this with the Metamath convention of naming "inference" versions of theorems: `[ax2i X Y Z]`.

9.3 Virtual namespaces

Sometimes it is necessary to create completely new theorem objects. For example, the deduction theorem converts a proof of ψ under the hypothesis ϕ to a proof of $\phi \to \psi$. To do this, it needs to recursively create deduction versions of theorems referenced by the proof.

There is a special macro `virtual-metath` for writing theorem-creating functions. A metatheorem created with `virtual-metath` should take exactly one argument that is a context, and also return a context. The results are memoized to avoid wasting space and time computing the same theorems again and again.

In addition to the standard way `[theorem ,argument]` of invoking metatheorems Bourbaki experimentally supports *virtual namespaces*. When a theorem reference `!/virtual!x!y!z` is parsed where `!/virtual` is created with `virtual-metath`, the remaining part `!x!y!z` is taken as a full name of a context. This context is given to `!/virtual` and the returned context is used instead of `z`. This way references can be printed to the theorems created on fly.

References

[1] Bourbaki: Elements of Mathematics. Theory of Sets, Chapter IV

[2] The Common Lisp HyperSpec, http://www.lisp.org/HyperSpec/FrontMatter/

[3] Megill: Metamath. Available at http://us.metamath.org/#book

[4] Mendelson: Introduction to Mathematical Logic, third edition, p. 44

[5] Megill: http://us.metamath.org/mpegif/mmset.html#distinct

[6] Lamport: How to Write a Proof, http://research.microsoft.com/users/lamport/pubs/pubs.html#lamport-how-to-write