

Klaus Grue, grue@diku.dk

Senior Software Engineer, Rovsing A/S

Rovsing does Independent Software Verification and Validation (ISVV) for space agencies and space companies (<http://www.rovsing.dk>)

Logiweb was developed at Dept.Comp.Sci., Univ.of Copenhagen (DIKU)

Logiweb is open source (GPL). See <http://logiweb.eu/>

Commented slides are at <http://logiweb.eu/grue> (click "MathWiki...")

Please use grue@diku.dk for contacting me concerning Logiweb. My Rovsing e-mail is for other purposes.

Objectives

- Accumulation of knowledge
- Standing on the shoulders of predecessors
- Notational freedom
- Foundational freedom
- Distribution over the Internet
- Readability/typography
- Accommodation (small as well as large pages)
- Scalability: run smoothly up to 10^{18} papers.
- Simplicity
- Extensibility
- Automatic verification

Accumulation of knowledge: Once a machine formalized development was published on Logiweb, the publication should remain accessible in unchanged form forever. This simply mimics what publishing houses and libraries provide for ordinary mathematics.

Standing on the shoulders of predecessors: When proving a theorem, users should be able to draw upon theorems proved by others. This simply mimics the way any mathematician works.

Notational freedom: Each user should be free to use any notation. That freedom should come without restricting the notational freedom of others. And differences in notation should be no obstacle when using the results of others.

Foundational freedom: Each user should be free to choose what mathematical foundation to work upon. Import of results from one foundation to another should be possible (but not necessarily trivial).

Distribution: Access to the work of others should happen transparently via the Internet.

Readability: Users should be able to publish articles on Logiweb which are as readable as any mathematics book one can pick from any library.

Accommodation: Users should be able to publish anything from short notes to multi-volume works that span thousands of pages.

Scalability: The system should allow an arbitrary number of submissions and should run smoothly up to 10^{18} papers.

Simplicity: The system should be so simple that its core could be implemented by a single person. And so simple that once it was implemented, a graduate computer science student could re-implement it in one year.

Extensibility: Once the core was implemented, users should be able to adapt the system without changing the core and should be able to publish adaptations on Logiweb itself to make the adaptations available to other users.

Verification: Logiweb should verify publications automatically.

Timeline



- 1975: First predecessor of verifier. Algebraic PA.
- 1984: Parser. FT.
- 1985: Second predecessor of verifier. Graduate course using verifier begins.
- 1992: Map theory.
- 1994: First year course based on verifier begins.
- 1996: Design of Logiweb starts.
- 2003: Machine verified exam on first year course.
- 2006: Logiweb 0.1.1: first beta release.
- 2008: Map theory 2.

Algebraic PA is Peano arithmetic expressed algebraically. That theory developed into FT (for “Formal Theory”) and then into map theory (which is λ -calculus plus a quantifier, which can simulate ZFC, and which has the same consistency power as ZFC. Logiweb can support these “exotic” theories as well as mainstream theories like FOL, PA, ZFC, NBG, and so on.

The parser from 1984 is essentially the frontend of the Logiweb compiler.

The graduate courses actually used the verifier: Students verified their theses using it. The first year courses only used the verifier indirectly: the textbook was written as a machine verifiable text and Logiweb was designed based on the experience gained by writing such a book. So the book was written first and the verifier afterwards.

Combinations

The number of combinations of size k from a set of size n is given by the binomial coefficient $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. A recursive definition of $\binom{n}{k}$ may be stated thus:

$$\left[\binom{n}{k} \doteq \text{if } k = 0 \text{ then } 1 \text{ else } \binom{n-1}{k-1} \cdot n \text{ div } k \right]$$

As an example, we have $\left[\binom{4}{2} = 6 \right]$.

The slide above is an example of what Logiweb can generate. Actually, this entire set of slides has been generated and verified by Logiweb.

The slide comprises some informal text which introduces the binomial coefficient $\binom{n}{k}$ and states that it equals $\frac{n!}{k!(n-k)!}$.

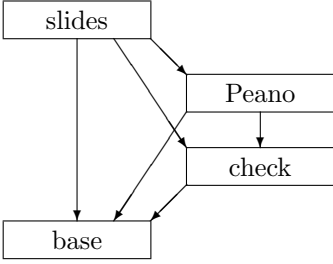
Then comes a formal definition of the binomial coefficient of form $\left[\binom{n}{k} \doteq \dots \right]$.

Logiweb takes note of such definitions.

Finally, the slide contains a test case $\left[\binom{4}{2} = 6 \right]$. Test cases can be recognized by the square brackets and the dot superscript. Logiweb has verified that test case using the given definition of the binomial coefficient.

To make a Logiweb page similar to the one above go to <http://logiweb.eu/> and run Tutorial T02.

References



The present slides reference three other pages named “base”, “check”, and “Peano”. The base page defines elementary constructs like $\lambda x.y$ and $[\dots \doteq \dots]$. The check page defines a proof checker. The Peano page defines Peano arithmetic.

On Logiweb, pages and references form a directed, acyclic graph. Each page can only reference previously published pages.

The Logiweb protocol is prepared for handling “back-references” such as references from pages in the past which state a theorem to pages in the future which prove the theorem. But such “back-references” will be implemented as “anchors” in the past pages together with references from the future pages to the anchors which have the special property that users can follow the references in the opposite direction of the direction they point.

The .pyk source of the present slides references the latest version of the check page. Each time the slides are re-published, the new slides will reference the latest version of the check page. If the check page is changed after publication of the slides, the slides will still reference the version of the check page which was “latest” at the time the slides were published.

A lemma in Peano arithmetic

We now state Lemma 3.21 of [1]:

PA lemma 3.21: $\forall x: 0 \cdot x = 0 \quad \square$

The slide above introduces a lemma. From the point of view of Logiweb, the slide defines the “statement” aspect of 3.21 to be $\text{PA} \vdash \forall x: 0 \cdot x = 0$. The PA construct stands for “Peano Arithmetic” and is a theory which is defined on a referenced page.

$\left[\binom{n}{k} \doteq \dots \right]$ on the previous slide defined the “value” aspect of the binomial coefficient. From the point of view of Logiweb, each definition sets a particular aspect of a particular construct to a particular term.

Logiweb allows users to define an indefinite number of aspects. Logiweb knows a few aspects like the “value” aspect in advance. Other aspects like the “statement” aspect are user defined.

A Proof



PA proof of 3.2l:

L01:	S7 \gg	$0 \cdot 0 = 0$;
L02:	Block \gg	Begin	;
L03:	Hypothesis \gg	$0 \cdot x = 0$;
L04:	S8 \gg	$0 \cdot x' = 0 \cdot x + 0$;
L05:	S5 \gg	$0 \cdot x + 0 = 0 \cdot x$;
L06:	$3.2c \supseteq L04 \supseteq L05 \gg$	$0 \cdot x' = 0 \cdot x$;
L07:	$3.2c \supseteq L06 \supseteq L03 \gg$	$0 \cdot x' = 0$;
L08:	Block \gg	End	;
L09:	Induction @ $x \triangleright L01 \triangleright L08 \gg$	$0 \cdot x = 0$;
L10:	Gen1 $\triangleright L09 \gg$	$\forall x: 0 \cdot x = 0$	□

The slide above proves the lemma on the previous slide. The proof above has been verified by Logiweb.

Before verification, the proof is macro expanded and tactic expanded. Furthermore, the proof is tactic expanded both at proof level and at proof line level.

At proof level, the proof is expanded according to the “tactic” aspect of PA. Hence, PA does not just define Peano arithmetic. It also defines how Peano proofs should be tactic expanded. It is the proof level tactic expander associated with PA which handles deduction by expanding begin-end blocks into axioms and inference rules. Theories different from Peano arithmetic may well have deduction theorems different from the deduction theorem of first order logic which makes it reasonable to associate deduction with the theory.

At proof line level, the unification tactic $\dots \gg \dots$ ensures that all axiom schemes and inference rules are instantiated suitably.

Hofstaedters MIU system

Axiom Double: $\Pi x: M \circ x \circ U = M \circ x \circ x \circ U \quad \square$

Axiom Add: $\Pi x: x \circ U = x \circ I \circ I \circ I \circ U \quad \square$

Axiom Assoc: $\Pi x, y, z: (x \circ y) \circ z = x \circ (y \circ z) \quad \square$

Rule Trans: $\Pi x, y, z: x = y \vdash y = z \vdash x = z \quad \square$

Rule Com: $\Pi x, y: x = y \vdash y = x \quad \square$

Theory MIU: Double \oplus Add \oplus Assoc \oplus Trans \oplus Com \square

The system above is not quite Hofstaedters MIU system, but it is close.

The sans serif variables $x, y,$ and so on are meta variables as opposed to object variables like $x, y,$ and so on.

$\Pi x, y, z: (x \circ y) \circ z = x \circ (y \circ z)$ is a meta statement which states that $(x \circ y) \circ z = x \circ (y \circ z)$ holds for all object terms $x, y,$ and $z.$

$\Pi x, y: x = y \vdash y = x$ states that $x = y$ infers $y = x$ for all object terms x and $y.$

The $\dots \vdash \dots$ construct is right associative such that $x = y \vdash y = z \vdash x = z$ means $x = y \vdash (y = z \vdash x = z).$

A MIU lemma and its proof



MIU lemma M-1-5: $M \circ I \circ U = M \circ I \circ I \circ I \circ I \circ I \circ U \quad \square$

MIU **proof of** M-1-5:

L01: Double \gg

$$M \circ I \circ U = M \circ I \circ I \circ U \quad ;$$

L02: Add \gg

$$M \circ I \circ I \circ U = M \circ I \circ I \circ I \circ I \circ U \quad ;$$

L03: Trans \triangleright L01 \triangleright L02 \gg

$$M \circ I \circ U = M \circ I \circ I \circ I \circ I \circ I \circ U \quad \square$$

Replacement (aka substitution of equals)

$[\text{Replace}(u, v, x, y) \doteq \lambda c. \text{Rep}(\lceil u \rceil, \lceil v \rceil, \lceil x \rceil, \lceil y \rceil)]$
 $[\text{Rep}(u, v, x, y) \doteq u \stackrel{t}{=} x \text{ and } v \stackrel{t}{=} y \text{ or } x \stackrel{r}{=} y \text{ and } \text{Rep}^*(u, v, x^t, y^t)]$
 $[\text{Rep}^*(u, v, x, y) \doteq x \text{ or } \text{Rep}(u, v, x^h, y^h) \text{ and } \text{Rep}^*(u, v, x^t, y^t)]$

Rule Replace: $\Pi u, v, x, y: \text{Replace}(u, v, x, y) \Vdash u = v \vdash x = y \square$

Theory MIU': $\text{MIU} \oplus \text{Replace} \square$

MIU' **lemma** M-2-4: $M \circ I \circ I \circ U = M \circ I \circ I \circ I \circ I \circ U \square$

Substitution of equals is missing in the MIU system so we extend the MIU system into MIU' by adding substitution of equals.

We represent substitution of equals by the Replace axiom scheme which claims $\Pi u, v, x, y: \text{Replace}(u, v, x, y) \Vdash u = v \vdash x = y$. Or, in words, for all terms u , v , x , and y we have that if the side condition $\text{Replace}(u, v, x, y)$ holds then $u = v$ infers $x = y$. The endorsement operator $\cdots \Vdash \cdots$ has the same priority and associativity as $\cdots \vdash \cdots$.

The side condition itself is macro defined. $\text{Replace}(u, v, x, y)$ macro expands into $\lambda c. \text{Rep}(\lceil u \rceil, \lceil v \rceil, \lceil x \rceil, \lceil y \rceil)$. During proof checking, c is instantiated to the environment of the proof, i.e. to a structure which contains all definitions of the current and all its transitively referenced pages. But the Replace side condition disregards c . The arguments to Replace are all quoted during macro expansion. Quoting results in something which resembles a Lisp S-expression.

The $\text{Rep}(u, v, x, y)$ construct is true if u and x are equal terms and v and y are equal terms. It is also true if x and y have equal roots (equal principal operators) and $\text{Rep}(u, v, x_i, y_i)$ is true for all subterms of x and y . x^h and x^t denote the head and tail of x .

The definition $[\text{Rep}(u, v, x, y) \doteq \cdots]$ macro expands into a value definition of form $[\text{Rep}(u, v, x, y) \doteq \cdots]$. But it does so in a way which hints that Logiweb should apply heavy optimization to the definition.

The definition of Replace is a bit naive. Normally one does not allow substitution of equals inside quotes and similar constructs. Knowing which constructs are quotes requires access to the environment c .

A MIU' proof



MIU' proof of M-2-4:

L01:	Assoc \gg	$(M \circ I) \circ I = M \circ (I \circ I)$;
L02:	Assoc \gg	$(M \circ I \circ I \circ I) \circ I = M \circ I \circ I \circ (I \circ I)$;
L03:	Replace \triangleright L01 \gg	$(M \circ I) \circ I \circ U = M \circ (I \circ I) \circ U$;
L04:	Double \gg	$M \circ (I \circ I) \circ U =$ $M \circ (I \circ I) \circ (I \circ I) \circ U$;
L05:	Com \triangleright (Replace \triangleright L01) \gg	$M \circ (I \circ I) \circ (I \circ I) \circ U =$ $(M \circ I) \circ I \circ (I \circ I) \circ U$;
L06:	Com \triangleright (Replace \triangleright L02) \gg	$M \circ I \circ I \circ (I \circ I) \circ U =$ $(M \circ I \circ I \circ I) \circ I \circ U$;
L07:	Trans \triangleright L03 \triangleright (Trans \triangleright L04 \triangleright (Trans \triangleright L05 \triangleright L06)) \gg	$M \circ I \circ I \circ U = M \circ I \circ I \circ I \circ I \circ U$	□

Running example

Define $|x| \doteq \mathbf{if } x \geq 0 \mathbf{ then } x \mathbf{ else } -x$ and $|x| \stackrel{\text{tex}}{\equiv} '\text{\texttt{vert}}'[x]'\text{\texttt{vert}}'$.

We shall use a very small page with the rendering above as running example for explaining the Logiweb compiler. The pyk source is given on the next slide.

```
PAGE abs
BIBLIOGRAPHY
"base" "http://logiweb.eu/logiweb/page/base/fixed/vector/page.lgw".
PREASSOCIATIVE
"base" base
"" | " |
PREASSOCIATIVE
"base" +
...
BODY text "page.tex" :
"\documentclass{article}\everymath{\rm}\begin{document}
Define "[[ value define | x | as if x >= 0 then x else - x end
define ]]" and "[[ tex define | x | as "\vert"[ x ]"\vert"
end define ]]".\end{document}" end text ,,
latex ( "page" ) ,, dvipdfm ( "page" )
```

Running example

```

... text "page.tex" :
"\documentclass{article}\everymath{\rm}\begin{document}
Define "[[ value define | x | as if x >= 0 then x else - x end define ]]"
and "[[ tex define | x | as "\vert"[ x ]"\vert" end define ]]" .
\end{document}" end text ,, latex ( "page" ) ,, dvipdfm ( "page" )

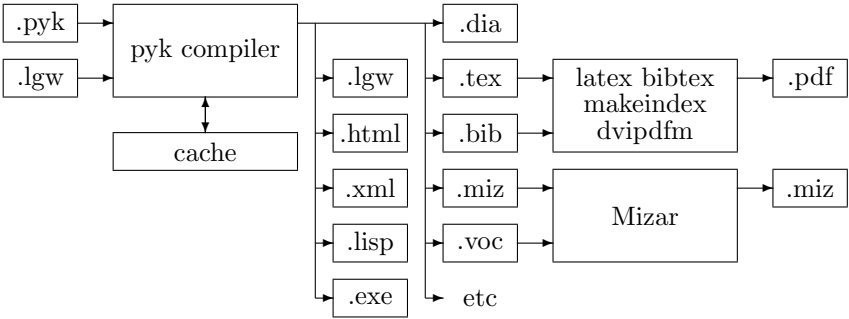
```

pyk compiler

```

Define [|x| ≐ if x ≥ 0 then x else -x] and [|x|  $\stackrel{\text{tex}}{=} \text{'\vert'[x]\vert'}$ ].

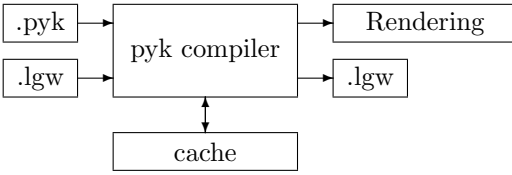
```



Users may express a Logiweb page as a `.pyk` source file. The `pyk` compiler can translate such a `.pyk` source file to a Logiweb `.lgw` file. The `pyk` compiler is also able to “render” the page in `.html`, `.xml`, `.lisp`, `.exe`, and many other formats (any other formats, actually). The `pyk` compiler is also able to invoke external programs like `latex`, `bibtex`, `makeindex`, `dvipdfm`, and `Mizar` on rendered files.

If the Logiweb page defined in the `.pyk` source file references other Logiweb pages, then it “loads” the `.lgw` files of those referenced pages. Loading a referenced file involves: Locating the `.lgw` file on the Internet using the Logiweb protocol, retrieving the file using `http`, unpacking the file, “understanding” the file, and verifying the file. Loading a referenced page involves loading of all pages referenced by that page, so one ends up loading all transitively referenced files.

Pyk compiler, simplified view



The “Rendering” covers files of type .html, .xml, .lisp, .exe, .dia (diagnose), .tex, .bib, .miz, .voc, .pdf, and so on which are not meant for internal consumption by Logiweb. The “Rendering” may be consumed by human readers or external systems like latex, bibtex, makeindex, dvi_{pdf}m, Mizar, and so on. In particular, a rendering may comprise executable files.

The “cache” is for internal Logiweb use. But it is easy to read by external systems and give easy access to the “innermost thoughts” of Logiweb.

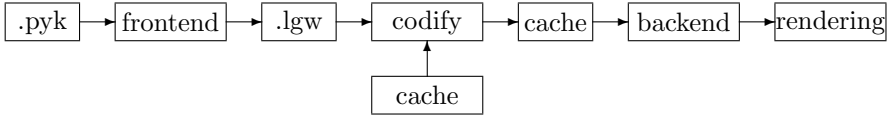
The cache is stored on disk and contains one file for each Logiweb page cached by the compiler. The file name is identical to the reference of the page.

Cache files are too big to be suitable for transmission over the Internet.

Having the .lgw file of a Logiweb page, one can re-generate the rendering and cache file. The .lgw files are rather compact and suited for transmission over the Internet.

From the point of view of the user, the pyk compiler takes a .pyk file as input, checks it, and delivers a rendering as output.

Structure of pyk compiler



The frontend translates the .pyk source file into a .lgw “byte vector”.

The codifier “understands” and verifies that .lgw vector. The result of that is a “cache”. The codifier loads the cache of each transitively referenced page. If some page is missing in the cache, the codifier instead locates the .lgw file on the Internet using the Logiweb protocol and the Logiweb servers. Then it “understands” and verifies the retrieved .lgw file and places it in the cache, and then proceeds.

The backend just takes the digested page and renders it.

Frontend applied to running example 1



The following slides describe what happens when the frontend is applied to the running example.



When the frontend is applied to the .pyk file, it produces an .lgw file.

An .lgw file comprises a bibliography, a dictionary, and a body.

Frontend applied to running example 2



When the frontend reads

`PAGE abs`

it adds “abs” as a production to its grammar and notes that “abs” denotes the page being defined.

Frontend applied to running example 3



When the frontend reads

BIBLIOGRAPHY

```
"base" "http://logiweb.eu/logiweb/page/base/fixed/vector/page.lgw".
```

it notes that reference number one of the running example is

```
017451CF6643931035C71796AC493D382EC8357EE9A390D5D6DBCDA0806
```

When the frontend reads the BIBLIOGRAPHY above, it retrieves the given page.lgw file using the http protocol and the given URL. page.lgw starts with a unique “reference” which is about 30 bytes long. The frontend parses page.lgw until it has found the end of the reference.

If a new version of the base page is published at the given url, then the new base page will get its own, unique reference. The reference above will continue to point to the old version of the base page.

Frontend applied to running example 4



When the frontend reads

```
PREASSOCIATIVE
```

```
"base" base
```

```
" | " |
```

it adds `base` and `| " |` to its grammar.

A production like `| " |` states that a vertical bar followed by one or more spaces followed by a subexpression followed by one or more spaces followed by a vertical bar is a legal grammatical construct. In general, double quotes are used as placeholders in productions. As an example, the if-then-else construct has the following grammar: `if " then " else "`

Each production starts with a page name enclosed in double quotes. The line `"base" base`

imports the `base` construct from the base page. The line

```
" | " |
```

imports the `| " |` construct from no page which is another way of stating that the construct is a new one which belongs to the present page.

The frontend assigns the same associativity to `base` and `| " |` and makes them both “preassociative” (i.e. “left associative” when writing left to right). The frontend silently assigns the same priority and associativity to the `abs` construct from Page 19.

When the frontend reads `"base" base` it “cheats” a little: it imports not only the `base` construct from the base page. It also imports all constructs from the base page which have the same priority as the `base` construct. There are hundreds of such constructs, so this convention saves the user a lot of typing. On the other hand, the frontend needs to “understand” the base page to know the priorities of all its constructs, so the frontend actually invokes the codifier on the base page. So codification of referenced pages actually occurs a little earlier than stated on page 17. But this is a messy detail.

Frontend applied to running example 5



When the frontend reads
PREASSOCIATIVE
"base" +"

it imports the "+" construct from the base page.

Tacitly, the frontend also imports "-", "0", "1", ..., "9" because they have the same priority as "+".

The "+" construct is “gluing” in that there is no space between the plus sign and the parameter.

The constructs "+", "-", "0", ..., "9" together with the constructs "0", ..., "9" are the constructs for making numerals such as -117. The numeral -117 is parsed as `-[1[1[7]]]` and macro expands into something whose value is -117.

Frontend applied to running example 6

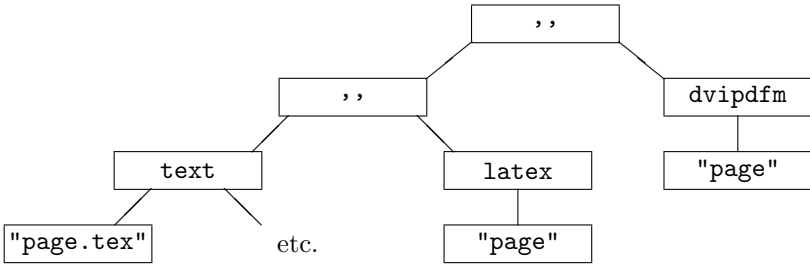


When the frontend reads the body

```
text "page.tex" :  
"\documentclass{article}\everymath{\rm}\begin{document}  
Define "[[ value define | x | as if x >= 0 then x else - x end define ]]"  
and "[[ tex define | x | as "\vert" [ x ] "\vert" end define ]]" .  
\end{document}" end text ,, latex ( "page" ) ,, dvipdfm ( "page" )
```

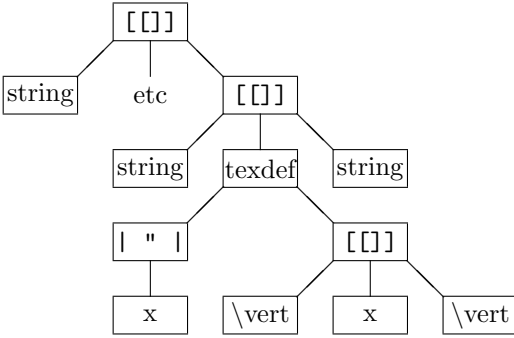
it parses it according to its grammar. The result is given on the next slide

Frontend applied to running example 7



The branch denoted “etc” continues next slide.

Frontend applied to running example 8



The boxes marked “string” denote various strings. The boxes marked `\\vert` denote the strings `"\\vert"`. The branch denoted “etc” denotes the parse tree of `|x| \doteq if $x \geq 0$ then x else $-x$` .

Frontend applied to running example 9



At the end, the frontend produces the following .lgw file:

```
Bibliography      01AC01EDD08426AF1E5AEOE5EB8785885E97C005CFEAEF
                  FCC0E5E2AC0806
                  017451CF6643931035C71796AC493D382EC8357EE9A390
                  D5D6DBCDA0806
Terminator        00
Dictionary        0101
Terminator        00
Body              Body in Polish prefix
```

The bibliography comprises two references. Reference 0 is the reference of the abs page itself and is generated by the frontend. Reference 1 is the reference of the **base** page. The presence of this reference in the bibliography ensures that it is recorded exactly which version of the base page is referenced.

The frontend assigns indexes to constructs, starting at zero. The abs construct gets index 0 and the | " | construct gets index 1.

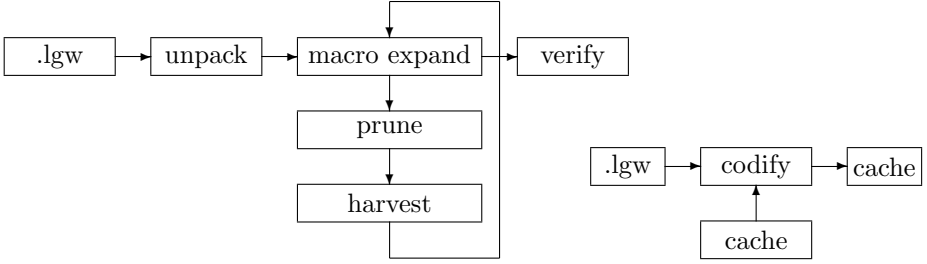
The names of the constructs are lost in translation, but the arities of each construct is recorded in the dictionary. The name of the page always gets index 0 and always has arity 0 so it is not included in the dictionary. Construct 1 has arity 1. That is why the dictionary contains the bytes 01 01. In general, the dictionary is a 00 terminated list of index/arity pairs.

Numbers below 128 are encoded in a single byte, numbers below 128^2 are encoded in two bytes, and so on. Numbers are encoded base 128. Bytes whose most significant bit is zero marks the end of a number.

The body contains strings and constructs. Each string is encoded as a 00 byte followed by the length of the string encoded base 128 followed by the bytes of the string. When strings are used for representing sequences of characters, Logiweb uses Logiweb Unicode UTF-8 encoding. That encoding differs from ordinary Unicode UTF-8 encoding in that codes 0..9 and 11..31 are illegal characters in Logiweb Unicode UTF-8. Logiweb uses code 10 for the newline character regardless of underlying operating system.

Each construct is represented by $r+ni$ encoded base 128. n is the length of the bibliography including reference 0 ($n = 2$ for the abs page). i is the index of the construct. r is the bibliographic entry (0 for the abs page, 1 for the base page).

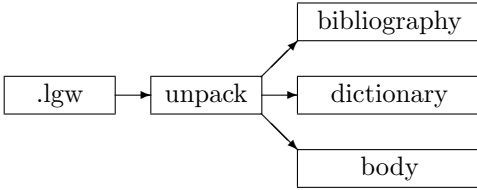
Codifier applied to running example 1



The codifier takes the .lgw byte vector as input and produces a cache.

The codifier comprises five processes: unpack, macro expand, prune, harvest, and verify. These processes all interact with the cache and accumulate their output in the cache.

Codifier applied to running example 2



The unpacker just reverses the encoding made by the frontend and extracts the bibliography, dictionary, and body from the .lgw file. The bibliography is stored at the “bibliography hook” of the cache entry for the abs page. The dictionary and body are stored in a similar way.

Unpacking of bibliography and dictionary is fixed by the Logiweb standard.

As soon as the bibliography is unpacked, the unpacker invokes the codifier recursively on all referenced pages. Hence, at the end of unpacking, all transitively referenced pages are in the cache. Loading is suppressed for pages which are already in the cache.

The codifier has both an internal cache and a cache on disk. The disk cache contains one file per page and persists from one invocation of the pyk compiler to the next. The internal cache only contains the pages transitively referenced from the page being translated.

By default, unpacking of the body is the reverse of the encoding done by the frontend. But reference 1 in the bibliography can specify an arbitrary function to do the unpacking of the body. In this way one can compress pages and specify in the unpacker how to decompress them. Or one can encrypt pages and specify in the unpacker how to decrypt them (using some public key decryption). Or one may decide to support formats different from Logiwebs .lgw format.

Codifier applied to running example 3

The macro expander expands the body

Define $[[x] \doteq \mathbf{if} \ x \geq 0 \ \mathbf{then} \ x \ \mathbf{else} \ -x]$ and $[[x] \stackrel{\text{tex}}{=} \text{'\vert'}[x]\text{'\vert'}]$.

to the expansion

Define $[[x] \stackrel{\text{val}}{\mapsto} \mathbf{if} \ x \geq 0 \ \mathbf{then} \ x \ \mathbf{else} \ -x]$ and $[[x] \stackrel{\text{tex}}{\mapsto} \text{'\vert'}[x]\text{'\vert'}]$.

The macro expander takes the body from the cache and stores the expansion on the “expansion” hook of the abs page in the cache.

By default, the body is macro expanded by applying the identity function to the body. In most cases, however, the user departs from that default.

In the abs example, reference 1 (the base page) defines a macro expander which is available at this point since reference 1 was loaded during unpacking. The pyk compiler simply applies the macro expander defined on the base page to the body. The macro expander defined on the base page allows the user to assign macro definitions to individual constructs, allows macro expansion to proceed recursively, and allows the user to obtain all sorts of effects. But any user can define and use a completely different macro expander if they want.

In the abs example, macro expansion reduces the $[\dots \doteq \dots]$ and $[\dots \stackrel{\text{tex}}{=} \dots]$ constructs to the ternary definition construct $[\dots \stackrel{\text{val}}{\mapsto} \dots]$. The ternary definition construct takes three arguments: left hand side, right hand side, and aspect.

Codifier applied to running example 4



The pruner traverses the output from the macro expander recursively and whenever it sees something malformed, it cuts it off and replaces it by something wellformed.

The user can use any function as macro expander, so the output from the macro expander needs not be well-formed. But if pruning terminates in finite time then the output from pruning is guaranteed to be well-formed. And if the input to the pruner is well-formed, the pruner does not change it.

Codifier applied to running example 5



The harvester traverses the expansion and collects all definitions.

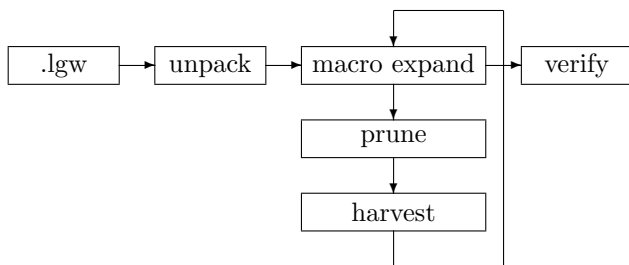
Among other, the harvester hangs

$[|x| \xrightarrow{\text{val}} \mathbf{if } x \geq 0 \mathbf{ then } x \mathbf{ else } -x]$

on the `value` branch of the `| "` | branch of the `codex` hook of the cache.

Codifier applied to running example 6

And then the body is macro expanded again! This continues until two consecutive expansions are equal.



After harvesting, the available macro definitions may have changed. So the result of macro expansion may be different from the result of the next macro expansion. But if two consecutive macro expansions give the same result then a fixed point has been reached, and this particular fixed point is then the “official” expansion of the body.

The base page is a bit complex. This is so among other because it defines its own macro expander. The base page must be macro expanded seven times before a fixed point is reached.

Macro expansion may proceed indefinitely without reaching a fixed point. That is not my problem. That is the users problem. If you ask for trouble in Logiweb, you get it.

But the pyk compiler has an option for halting macro expansion after a prescribed number of iterations. That allows the user to debug the problem.

Note that the search of a fixed point is very similar to the way cross references are handled in \TeX : Just rerun \TeX until the cross references change no more.

Codifier applied to running example 7



When macro expansion ends, the page is verified.

By default, any page is correct. In most cases, however, the user departs from that default.

In the abs example, reference 1 (the base page) defines a verifier which is available at this point since reference 1 was loaded during unpacking. The pyk compiler simply applies the verifier defined on the base page to the cache and the expansion and hangs the result on the `diagnose` hook of the cache. If the `diagnose` is empty then the page is correct.

The verifier defined on the base page traverses the expansion and verifies all test cases found. The verifier defined on the check page does the same but also traverses the cache for proofs and verifies all proofs.

Nothing prevents users from defining foolish verifiers.

The proof verifier on the check page only trusts itself in the sense that whenever it sees a proof which references a lemma on some other page, then the verifier looks up the verifier of the other page and checks that it has itself verified that page.

When the thing hanged on the `diagnose` hook is non-empty, it is pruned before entry into the cache. This ensures that the `diagnose` is a well-formed term which can be rendered.

Backend applied to running example 1



By default, the backend renders the body according “tex” definitions.

The default is sufficient for quite many cases. But one can override the default by defining a custom renderer on the first reference of the page.

The output from rendering is “executed”. By default, execution may invoke e.g. latex, bibtex, makeindex and dvi_{pdf}m. But one can change that in a configuration file and allow access to e.g. Mizar.

The “tex” definitions define how constructs normally look. One may also state “tex name” definitions which define how constructs look in special cases. Among other, the tex name definition is used for constructs when they appear in the left hand side of a definition. As an example, consider the `newline` "

construct. The “tex” definition renders the construct as “`\newline`” whereas the “tex name” definition renders it as “`newline`”. This makes the `newline` construct visible when it occurs in the left hand side of definitions. In other positions the `newline` construct just forces a line break.

Backend applied to running example 2



Furthermore, using an “execute” definition one can ask the pyk compiler to generate “Logiweb machines”. Logiweb machines are Turing complete machines with general input/output capabilities, interrupt handling, distinction between supervisor and user mode, and non-preemptive scheduling.

Logiweb machines can implement anything from “hello world” programs to operating systems. Logiweb machines can be implemented such that they can run hard real time, safety critical software.

Work is in progress to implement Logiweb as a Logiweb machine to get rid of the dependency of CLISP

Customization



The user can customize the following on Logiweb:

Unpacker

Macro expander

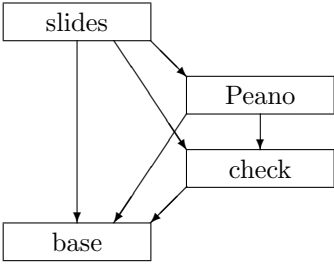
Verifier

Renderer

The macro expander and verifier defined on the base and check pages respectively in turn give lots of customization options to the user.

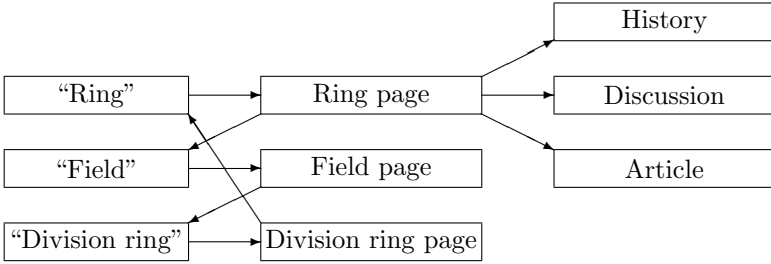
Among other, the macro expander on the base page provides a Turing complete macro expansion facility and the verifier provides Turing complete facilities for writing side conditions and proof tactics.

References



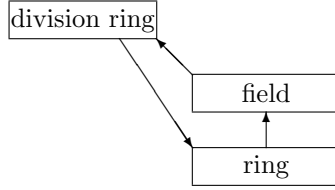
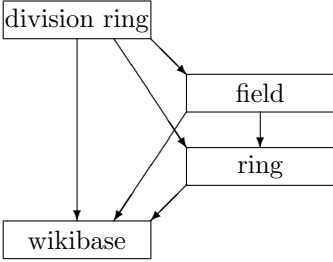
Recall that on Logiweb, pages and references form a directed, acyclic graph. Each page can only reference previously published pages.

But this only means that the Logiweb references form an acyclic graph. One can freely embed http references e.g. in the body of Logiweb pages and such http references need not form an acyclic graph.



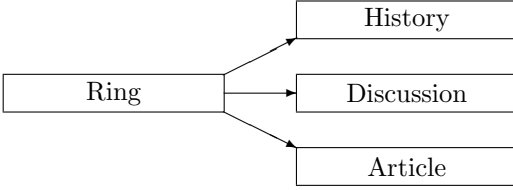
Wikipedia breaks circularity by letting pages contain “labels” which point back to the latest version of the referenced page.

Logiweb wiki?



In Logiweb there are no problems having both the structures above. The structure to the left is the “formal” structure which is reflected in the bibliographies of pages and which is used for verification. The structure to the right can be implemented by including http-references in page bodies.

Logiweb wiki?



If one disables garbage collection then Logiweb will automatically preserve the entire history. The only problem is that it may take up a lot of disk space. Furthermore, one may be interested in a change log. And that is missing.

Furthermore, Logiweb has no blogging interface.

Logiweb wiki?

The following is missing to use Logiweb for a wiki:

- A blog interface
- Recording of a changelog (who changed what how?)
- User login
- Possibly a better way of handling cross page operator precedence
- Possibly a nicer web interface

The question of the handling of operator precedence is particularly difficult. The current solution is to request each page to define its own operator precedence hierarchy. Another could be to assign fixed priorities to operators.

The priority scheme of e.g. Prolog where operators get a precedence in the range from 0 to 255 would be difficult to use. Instead one could define a priority as a sequence of integers and then use lexicographic ordering so that e.g.

$$\langle 1, 3 \rangle < \langle 2, -5 \rangle < \langle 2 \rangle < \langle 2, 4 \rangle$$

Programming 1

Logiweb uses lambda calculus. Example:

$[\lambda x.y \bowtie \text{'lambda'}]$

$[x \text{ ' } y \bowtie \text{'apply'}]$

$[\top \bowtie \text{'true'}]$

[If x then y else z \bowtie 'if']

$[x \text{ LazyPair } y \xrightarrow{\text{val}} \lambda z.\mathbf{If } z \text{ then } x \text{ else } y]$

$[\mathbf{F} \xrightarrow{\text{val}} \top \text{ LazyPair } \top]$

$[x \text{ Head} \xrightarrow{\text{val}} x \text{ ' } \top]$

$[x \text{ Tail} \xrightarrow{\text{val}} x \text{ ' } \mathbf{F}]$

The example above starts from scratch by “proclaiming” $\lambda x.y$ to denote lambda abstraction. Such proclamations typically appear on base pages. Proclamations ensures notational freedom even for fundamental constructs.

The example defines a LazyPair operator and its associated Head and Tail operators.

The example also “introduces” falsehood \mathbf{F} . An introduction is almost the same as a definition. But an introduction suggests to the pyk compiler that this particular construct is something the compiler should know. The pyk compiler then scans definitions of all constructs it knows, and when it finds one identical to the definition of \mathbf{F} above (modulo naming of bound variables and modulo naming of auxiliary functions) then the pyk compiler knows that this particular construct denotes falsehood.

Proclamations are used for fundamental constructs.

Introductions are used for constructs which in principle can be defined from the fundamental ones but which typically are implemented somehow directly in the pyk compiler.

Definitions are used for other constructs.

Interfacing 1

One can interface Logiweb to e.g. Mizar in several ways.

- One may list Mizar as a function callable from Logiweb. In that case Logiweb can be used to generate .miz files which are passed to Mizar. Verification of the .miz file occurs outside Logiwebs version control. If a new version of Mizar is issued, a once correct .miz file may become incorrect. For that reason, Logiweb can generate the .miz file and invoke Mizar on it, but cannot record that the .miz file is correct (since Logiweb only records “eternal truth”).
-

Interfacing 2



- One could publish the Pascal sources of Mizar on Logiweb and record the Pascal compiler as callable from Logiweb. Then Logiweb can rebuild a particular version of Mizar and invoke that particular version on the .miz file. In this case Logiweb still cannot record the outcome of the process since the Pascal compiler is outside Logiweb version control.
-

Interfacing 3

- One may implement a Pascal to lambda calculus compiler in lambda calculus and run the Mizar Pascal source through that compiler. This is not as silly as it may sound since one can arrange various backstage optimizations in the pyk compiler. This would bring Mizar under Logiweb version control. Actually it would allow to publish new versions of Mizar on Logiweb.
 - One may port Mizar to lambda calculus. But that probably requires a rather dedicated programmer.
-

Space applications

Logiweb is being adapted for hard real time, safety critical software. Until further, it has been proposed for use in connection with two projects:

- Autonomous Image Processing Chain (AIPC)
 - Crew Space Transportation System (CSTS)
-

Closing remarks 1

Some purposes of Logiweb are

- to support formalization of mathematics and
 - to give maximal freedom to each user
 - without restricting the freedom of other users
 - while keeping interoperability
 - and supporting interfaces to other systems
-

Closing remarks 2



There are tutorials at <http://logiweb.eu/>

At <http://logiweb.eu/> you can try Logiweb in your browser without installing software

You can download the system from <http://logiweb.eu/>

Logiweb runs on CLISP and Linux. Work is in progress on porting to C.

A full Logiweb installation requires an http server (preferably Apache) to be installed.

References

- [1] E. Mendelson. *Introduction to Mathematical Logic*. Wadsworth and Brooks, 3. edition, 1987.