

Logiweb

Klaus Grue¹

*Department of Computer Science (DIKU)
University of Copenhagen
Copenhagen, Denmark*

Abstract

This paper presents a software system with the preliminary name “Logiweb” which is an attempt to design a simple, coherent, all-embracing system for archiving, retrieval, transmission, distribution, verification, and presentation of mathematics. The design is made from scratch but depends heavily on experience from existing systems and the current stage of available technology.

Logiweb ensures that pages, once published, remain unchanged in the future so that it is safe to refer to Logiweb pages from mathematical papers. In this, Logiweb supplements the World Wide Web which is more suited for mathematics in flux.

Key words: Proof checker, archival of mathematics, retrieval of mathematics, notational freedom, protocols, mathematical knowledge management

1 Introduction

Logiweb is a web-like system that allows logicians to web-publish papers with high typographic quality and high human readability which are also machine verifiable. Among other, Logiweb allows papers to contain definitions of formal theories, definitions of new constructs, programs, lemmas, conjectures, and proofs. Furthermore, Logiweb allows papers to refer to each other across the Internet, and allows proof checking of proofs that span several papers that reside different places in the world. As an example, a lemma in one paper may refer to a construct which is defined in another paper situated elsewhere, in which case the proof checker must access both papers to establish the correctness of the proof.

Among other, Logiweb provides a medium for archived mathematics. It may be used as it is, or it may run silently and transparently underneath other systems like Mizar [12,16]. In contrast, the World Wide Web, which supports

¹ Email: grue@diku.dk

mathematics through MathML and OMDoc [10,11], is a medium suited for information in flux.

Logiweb gives complete notational freedom to its users as well as complete freedom to choose any axiomatic theory (e.g. ZFC) as basis for their work. Logiweb also allows different notational systems and theories to co-exist and interact smoothly.

Logiweb was originally designed to support Map Theory [2,7,15,17] which has the same power as ZFC but relies on very different foundations in that, e.g., it relies on λ -calculus *instead* of first order predicate calculus. The tie to Map Theory has been broken, however, in the sense that Logiweb supports all axiomatic theories equally well.

The ability of Logiweb to span from a predicate calculus based theory like ZFC to a λ -calculus based theory like Map Theory guarantees that Logiweb puts no restrictions on the logic. The absence of such restrictions of course makes it impossible to supply a code-from-theorems extraction facility like `term_of` of Nuprl [4], but functions for manipulation of theorems and proofs of individual theories are expressible in the programming language of Logiweb. The programming language of Logiweb is the programming language embedded in Map Theory which makes it possible to use Map Theory to reason *about* Logiweb.

Logiweb allows individuals to publish papers, but also allows journals to exist. Technically, a journal issue will just be a Logiweb page with references to accepted papers. Socially, however, journal editors are likely to enforce guidelines on typography, style, use of symbols and so on on their authors to ensure coherence within their journal and coherence with standard literature. Logiweb itself is neutral to such requirements, and guidelines for one journal does not restrict other journals and does not restrict the notational freedom of authors who reference the journals.

1.1 *Development history*

Logiweb draws heavily from the experience of other proof systems. Among many sources, Logiweb is based on proof checkers implemented by the author 1985-1992.

One goal of Logiweb was to design a simple proof system which allows to cope with the complexity of mathematical textbooks. To ensure that the system can cope with the complexity of a full, mathematical textbook in a human readable style, two books [8,9] have been developed 1992-2001 to test the system.

Reference [8] is a discrete math book for first year university students and is of interest here because it has been possible to test the human readability of the book in practice.

Reference [9] is a treatise on Map Theory and is of interest here because it contains a substantial proof (a proof of the consistency of ZFC expressed

in Map Theory) that can stress test Logiweb. To allow comparison with other proof systems and to ensure correctness, [9] has been ported by hand to Isabelle [13,14,15].

One purpose of writing [8,9] was to identify syntactic and semantic features a mathematical knowledge management system has to have in order to support textbooks and mathematical papers. Among other, the macro facilities presented in Section 6 were developed to cope with such demands.

Logiweb is currently used by students who use the parts of Logiweb already implemented and provide feedback. An experimental version for a broader audience is due for the beginning of next year.

1.2 Overview of the paper

The sections of the paper start with short introductions which contain remarks, motivations, and comparisons with some literature relevant to the given section.

Section 2 gives short examples on what Logiweb looks like to the user. For 500 and 300 page examples, consult [8] and [9], respectively.

Section 3 describes how Logiweb ensures that pages, once published, remain unchanged and available in the future so that it is safe to refer to Logiweb pages from mathematical papers.

Section 4 describes the main data structures in Logiweb: (1) Parsetrees that describe the structure of pages and are the entities stored by Logiweb. (2) Codices which are extracted from parsetrees and represent pages in a more machine understandable, data-base like format. (3) Symbols that represent concepts. (4) Aspects, which allow symbols to have different meanings in different contexts.

Section 5 describes how Logiweb ensures notational freedom, foundational freedom, and independence of individual authors and at the same time supports interdependence in allowing authors to refer to one another and use the results of each other.

Section 6 describes the macro facility of Logiweb which allows to keep the de Bruijn factor [5] low and supports pages in being human and machine readable at the same time.

Section 7 describes the programming language of Logiweb. The programming language of Logiweb is the programming language embedded in Map Theory [2,7,15,17] which allows Logiweb to use Map Theory (which has ZFC power) for reasoning *about* Logiweb.

Section 8 describes how Logiweb does verification of pages and describes how to change the default correctness criterion of Logiweb to e.g. the Mizar [12,16] correctness criterion, thereby allowing to embed other systems in Logiweb.

Section 9 describes how to bootstrap the system which is non-trivial in a system which offers complete notational freedom and, hence, for which one cannot rely on any construct having any, predefined syntax.

Appendix A describes the byte level protocol used among Logiweb servers for storing and retrieving uninterpreted Logiweb pages.

Appendix B describes the protocol used by Logiweb browsers for interpreting Logiweb pages.

2 Getting started

From the point of view of an implementor, Logiweb is a protocol, but from the point of view of a user, Logiweb is a collection of “Logiweb pages”, where a page may be anything from a single line of text to a full textbook like [8]. To use Logiweb, the user needs a Logiweb browser plus Internet access to Logiweb servers that store pages. The following sections describe how two persons, B and C, who work at different sites, may cooperate using Logiweb.

2.1 Definitions and statements

As a minimalistic example, suppose Person B wants to define $f(x) \doteq x^2 - 1$ on Logiweb. B may do so using Logiweb by publishing a (very short) page like this:

The function $f(x)$ is defined by $[f(x) \doteq x^2 - 1]$.

The brackets guide correctness checks: Text in brackets should be read by a proof checker. Text outside brackets is commentary.

Person C may find B’s page somehow, and may want to state the observation that $f(10) = 99$. C may do so by publishing another page:

We have $[f(10) = 99]$.

Every page contains a bibliography (which needs not be visible). Above, the latter page must refer to the former to indicate that the $f(x)$ on the latter page is the one defined on the former. In turn, the former page must refer to pages that define x^y , $x - y$, $x \doteq y$, and numerals.

2.2 Correctness

A page may be “correct” or “erroneous”. A Logiweb browser must be able to “verify” a page (i.e. check that it is correct). The pages above are correct, the first one because its definition is acceptable and the second because its claim is true. The two pages above reside different places in the world. To verify the latter page, a Logiweb browser must access both pages to get the definition as well as the claim.

The pages are correct according to the Logiweb default criterion. A page may override the default by specifying an alternative criterion. As an example, one may put a Mizar [12,16] paper on Logiweb and specify that it is correct according to the Mizar criterion. Details on correctness criteria are given later.

According to the default criterion,

Define $[f(x) \doteq x^2 - 1]$ and $[f(x) \doteq x^2 - 2]$.

and

We have $[f(10) = 100]$.

are erroneous, the first because the two definitions contradict each other and the second because its claim is false.

2.3 Lemmas and proofs

The default correctness criterion allows to define rules and theories, and to state and prove lemmas. Suppose Person B publishes the following page:

[Rule P^0 : $(x) x + 0 = x$]
 [Rule P' : $(x, y) x + y' = (x + y)'$]
 [Rule Transitivity: $(x, y, z) x = y \vdash x = z \vdash y = z$]
 [Theory S_+ : $P^0 \oplus P' \oplus$ Transitivity]
 [S_+ lemma L2.3.1: $x = x + 0$]

The page defines P^0 to be the rule that $x + 0 = x$ for all terms x , P' to be the rule that $x + y' = (x + y)'$ for all terms x and y , and Transitivity to be the rule that if $x = y$ and $x = z$ then $y = z$ for all terms x , y , and z . Then the page defines S_+ to be the theory that has P^0 , P' , and Transitivity as axioms and inference rules and states as Lemma L2.3.1 that $x = x + 0$ is a theorem of S_+ . In the last line, x serves as a concrete variable; in the three first it serves as a meta-variable.

Then Person C may decide to prove $x = x + 0$ on yet another page:

[Proof of L2.3.1:
 L1: $P^0 \gg x + 0 = x$;
 L2: $P^0 \gg x + 0 + 0 = x + 0$;
 L3: Transitivity \triangleright L2 \triangleright L2 $\gg x + 0 = x + 0$;
 L4: Transitivity \triangleright L1 \triangleright L3 $\gg x = x + 0$]

Again, the two pages reside two different places in the world, and checking the second page requires access to both pages.

As a technical remark, the first page is correct according to the default criterion of “local correctness” which merely requires proofs to be correct but does not require all lemmas to have a proof. The user may choose a more stringent “global correctness” criterion, e.g. that all lemmas have proofs and all proofs use lemmas in a non-circular fashion.

3 Stability

When an author refers to a paper, the author typically expects the referenced paper to remain firm (i.e. unchanged) and available in the future. This section describes how Logiweb fulfills the stability requirements of firmness and availability.

3.1 Firmness

When a page is published, a “Logiweb reference” is assigned to the page. A reference is a natural number with about 200 bits and is made up of a RIPEMD-160 [6] hash key, a time stamp, and a version number, but references are not seen by the user. Now assume that the first two pages above get references $9 \cdots 91$ and $9 \cdots 92$ when published by Person B and C, respectively:

The function $f(x)$ is defined by $[f(x) \doteq x^2 - 1]$.

$9 \cdots 91$ (B’s page)

We have $[f(10) = 99]$.

$9 \cdots 92$ (C’s page)

At the time of publication, C’s page is correct, but if B could change $[f(x) \doteq x^2 - 1]$ to e.g. $[f(x) \doteq x^2 - 2]$, then C’s page would become erroneous.

On Logiweb, however, it is impossible to change a page once it is published. Person B can of course publish a page that says $[f(x) \doteq x^2 - 2]$, but then that page would get a new reference such as $9 \cdots 93$, and page $9 \cdots 92$ would still refer to page $9 \cdots 91$.

The reference of a page is based on a RIPEMD-160 hash code that is computed on basis of all of the contents of the page. As long as RIPEMD-160 stands up to collision attacks, even malicious users cannot change a Logiweb page once it is published. And no two users are able to publish different pages that get the same reference. The task of the Logiweb servers is to maintain a global hash table that translates references to pages.

As mentioned in the abstract, Logiweb is a medium for archived mathematics. In other words, Logiweb is accumulative in the sense that once a page is put on Logiweb, it is impossible to change it. This accumulative nature is

enforced by RIPEMD-160.

3.2 Availability

Even if Logiweb is accumulative, there may still be occasions where an author is no longer willing to contribute disk space to some old version of some old paper. When Person B above publishes page 9...91, the page will physically reside on B's computer. B may, however, delete page 9...91 which leaves C's page 9...92 meaningless.

By publishing on Logiweb, however, Person B implicitly grants copyright permission to other Logiweb servers to make verbatim copies of the page. In particular, person C's browser may silently tell C's Logiweb server to load and resubmit page 9...91 on C's computer so that page 9...91 remains available on Logiweb even if B deletes it on B's computer.

A page disappears from Logiweb when the last copy of the page disappears. When a page disappears, it may start an avalanche since all pages that reference the page, directly or indirectly, become incomplete. When a browser loads a page, it must also load all pages referenced, directly or indirectly, and the browser should report the page non-existent if any of these transitively referenced pages is missing. Hence, when the last copy of a page disappears, all its dependents effectively disappear as well.

For that reason, when an author publishes a page, it would be good practice for the author to secure a copy of all transitively referenced pages.

4 Data structures

The main data structures of Logiweb are *parsetrees* that are made up of *symbols*, and *codices* that associate *aspects* to symbols.

Parsetrees define the structure of a page and resemble expression trees in content encoding in MathML [11] and correspond to the tree model in OMDoc [10].

Codices represent the mathematical contents of a page in a more database-like structure which interfaces well with the Logiweb checking machinery and programming language. Logiweb codices contain mathematical semantics in a rather clean form as opposed to the more syntactical database-like structures of OMDoc ([10], p.65).

4.1 Symbols

The definition $f(x) \doteq x^2 - 1$ on page 9...91 above defines the value of $f(x)$ to be $x^2 - 1$.

To Logiweb, f is a "Logiweb symbol". A Logiweb symbol is a pair $\langle r, i \rangle$ of natural numbers where r is the reference of the home page of the symbol and i is a natural number that identifies the symbol within that page. We shall

refer to r and i as the “reference” and “index” of $\langle r, i \rangle$, respectively. As an example, f could be $\langle 9 \cdots 91, 1 \rangle$.

4.2 Parsetrees

Every Logiweb page contains a bibliography, a “parsetree”, and an “arity table”. Parsetrees have the following syntax (which resembles that of Lisp S-expressions):

$$\text{Parsetree} ::= \langle \text{Symbol}\{, \text{Parsetree}\}^* \rangle$$

Normally, a parsetree is the output of a parser which converts a human readable format to a machine friendly form. In Logiweb, the approach is opposite: The parsetree is the entity stored in the system, and human readable forms are derived from it.

The arity table assigns arities to the symbols of the page, i.e. a page with reference r defines the arities of the symbols of form $\langle r, i \rangle$. Logiweb forces all parsetrees to have form

$$\langle s, p_1, \dots, p_{\text{arity}(s)} \rangle$$

where $\text{arity}(s)$ denotes the arity of s .

4.3 Aspects

To Logiweb, $f(x) \doteq x^2 - 1$ is shorthand for

$$\text{Define}(\text{Value}, f(x), x^2 - 1)$$

which states that the “Value aspect” of $f(x)$ is $x^2 - 1$. In general, $\text{Define}(x, y, z)$ defines the x -aspect of y . As another example,

$$\text{Define}(\text{T}_{\text{E}}\text{X}, f(x), \text{small-f} \cdot \text{left-parenthesis} \cdot x \cdot \text{right-parenthesis})$$

defines the “T_EX aspect” of $f(x)$. To a Logiweb browser, $f(x)$ is a parsetree of form

$$\langle \langle 9 \cdots 91, 1 \rangle, x \rangle$$

The T_EX aspect tells the browser how to render $f(x)$ in T_EX. Other aspects may tell how to render $f(x)$ in e.g. MathML.

Page 9 \cdots 91 in the example has been simplified by omitting a T_EX definition of $f(x)$. In general, pages that introduce new symbols need an appendix that defines what the new symbols look like in various formats and possibly other things like the priority and associativity of the symbols.

To Logiweb,

Rule P^0 : $(x) x + 0 = x$,

Theory S_+ : $P^0 \oplus P' \oplus \text{Transitivity}$,

S_+ lemma L2.3.1: $x = x + 0$, and

Proof of L2.3.1: \cdots

are shorthand for

Define(Rule, P^0 , $(x) x + 0 = x$),
 Define(Theory, S_+ , $P^0 \oplus P' \oplus \text{Transitivity}$),
 Define(Lemma, L2.3.1, $S_+ \Vdash : x = x + 0$), and
 Define(Proof, L2.3.1, \dots)

respectively. Above, $x \Vdash y$ denotes the lemma that y is provable in the theory x .

Formally, an aspect is a symbol. To make a construct like Value denote the Value aspect, one must define the Message aspect of Value to be the symbol that represents the Value aspect.

As another example, one must also arrange the Message aspect of the Message construct to be the symbol that represents the Message aspect. How to do that is explained in Section 9 which covers bootstrapping.

4.4 Codices

A Logiweb browser maintains a ‘‘Logiweb codex’’ which stores the definitions that the browser encounters. Formally, a codex is a sparse five-dimensional array C (c.f. Section B.6) for which

$$C[p_r][s_r][s_i][a_r][a_i]$$

denotes the $\langle a_r, a_i \rangle$ -aspect of symbol $\langle s_r, s_i \rangle$ as defined on page $\langle p_r \rangle$.

As an example, when a browser loads

The function $f(x)$ is defined by $[f(x) \doteq x^2 - 1]$.

9 \dots 91 (B’s page)

then it notes in its codex that page 9 \dots 91 defines the Value aspect of symbol $\langle 9 \dots 91, 1 \rangle$ by the parsetree $\text{Define}(\text{Value}, f(x), x^2 - 1)$. When a browser loads

[Rule P^0 : $(x) x + 0 = x$]
 [Rule P' : $(x, y) x + y' = (x + y)'$]
 [Rule Transitivity: $(x, y, z) x = y \vdash x = z \vdash y = z$]
 [Theory S_+ : $P^0 \oplus P' \oplus \text{Transitivity}$]
 [S_+ lemma L2.3.1: $x = x + 0$]

9 \dots 94 (B’s page)

then it notes that page 9 \dots 94 defines the Rule aspect of symbol $\langle 9 \dots 94, 1 \rangle$ to be $\text{Define}(\text{Rule}, P^0, (x) x + 0 = x)$ and similar for symbol $\langle 9 \dots 94, 2 \rangle$ and $\langle 9 \dots 94, 3 \rangle$. Then it notes that the Theory aspect of symbol $\langle 9 \dots 94, 4 \rangle$ is

Define(Theory, S_+ , $P^0 \oplus P' \oplus$ Transitivity and the Lemma aspect of $\langle 9 \dots 94, 5 \rangle$ is Define(Lemma, L2.3.1, $x = x + 0$. When a browser loads

```
[Proof of L2.3.1:
L1:  $P^0 \gg$                                  $x + 0 = x$            ;
L2:  $P^0 \gg$                                  $x + 0 + 0 = x + 0$  ;
L3: Transitivity  $\triangleright$  L2  $\triangleright$  L2  $\gg$   $x + 0 = x + 0$    ;
L4: Transitivity  $\triangleright$  L1  $\triangleright$  L3  $\gg$   $x = x + 0$            ]
```

9 \dots 95 (C's page)

it notes that page 9 \dots 95 defines the Proof aspect of symbol $\langle 9 \dots 94, 5 \rangle$ to be the parsetree after the colon.

5 Independence

Logiweb is a medium that allows independent authors to publish interdependent pages. This section gives two examples of how the structure of the codex allows independence and interdependency to coexist.

5.1 Notational freedom

As mentioned, page 9 \dots 91 missed a definition of the \TeX aspect of $f(x)$. A more complete page would be:

```
The function  $f(x)$  is defined by [ $f(x) \doteq x^2 - 1$ ].
[Define( $\text{\TeX}$ ,  $f(x)$ , small-f · left-parenthesis ·  $x$  · right-parenthesis]
```

9 \dots 91 (B's page)

If Person C wants to use $f(x)$ but has reserved f for some other purpose, then C may rename f into g thus:

```
We have [ $g(10) = 99$ ].
[Define( $\text{\TeX}$ ,  $g(x)$ , small-g · left-parenthesis ·  $x$  · right-parenthesis]
```

9 \dots 92 (C's page)

Above, $g(x)$ is symbol $\langle 9 \dots 91, 1 \rangle$ in disguise. The pages above may confuse a human reader since C has been too lazy to make a comment that $g(x)$ on his page is the same as $f(x)$ and B's page. But a browser will just note that page 9 \dots 91 and 9 \dots 92 define the \TeX aspect of $\langle 9 \dots 91, 1 \rangle$ to be two different things and will use the \TeX aspect of page 9 \dots 91 and 9 \dots 92 when displaying page 9 \dots 91 and 9 \dots 92, respectively.

Hence, the notational freedom of B to call a function $f(x)$ does not restrict the notational freedom of C to rename it to $g(x)$. This is important because

world wide agreement on notation would require all mathematicians to use unreasonably long names for their concepts. Logiweb enforces world wide agreement on symbols in the sense that every symbol contains a page reference that RIPEMD-160 forces to be world wide unique, but that is detached from the human readable expression of symbols.

5.2 *Alternative proofs*

Section 4.4 gave an example where page $r = 9 \cdots 95$ defined the aspect $a =$ “Proof” of symbol $s = \langle 9 \cdots 94, 1 \rangle$. In general, a codex is an associative structure which associates values to reference-symbol-aspect triples $\langle r, s, a \rangle$.

Different pages may define the Proof aspect of symbol $\langle 9 \cdots 94, 1 \rangle$ differently and, hence, codices can cope with the situation where different authors publish different proofs for the same lemma.

6 Macros

To keep the de Bruijn factor [5] low, Logiweb includes a simple, yet powerful macro facility. The macro facility allows authors to write pages in a style that is appealing to the human reader but still macroreduces into a more machine understandable form.

6.1 *Macro definitions*

Logiweb has a default macro facility. Like anything else in Logiweb, A page may override the default by specifying an alternative.

As an example of use, consider the following definition:

$$\text{Define}(\text{Macro}, h(x), h(x) \dot{\rightarrow} x^2 - 1)$$

The definition above states that $h(x)$ is shorthand for $x^2 - 1$. $f(10)$ and $h(10)$ both evaluate to 99, but if $h(10)$ occurs in a proof, then it is expanded to $10^2 - 1$ before proof checking.

As a more complex example, suppose $x :: y$ denotes the pair of x and y and that $\langle \rangle$ denotes the empty tuple. Furthermore suppose $\langle x \rangle$ and x, y is a unary and a binary construct, respectively, and consider the following macro definition:

$$\text{Define}(\text{Macro}, \langle x \rangle, (u) (v) \langle u, v \rangle \dot{\rightarrow} u :: \langle v \rangle \ddot{\rightarrow} \langle x \rangle \dot{\rightarrow} x :: \langle \rangle)$$

The definition states that $\langle x \rangle$ should be macroreduced by $\langle u, v \rangle \dot{\rightarrow} u :: \langle v \rangle$ if the parsetree x has a comma-operator in its root and should reduce to $x :: \langle \rangle$ otherwise. As an example, $\langle 1, 2, 3 \rangle$ (where commas are right associative) would

macro reduce to $1 :: 2 :: 3 :: \langle \rangle$ as follows:

$$\begin{aligned} \langle 1, 2, 3 \rangle &\quad \ddot{\rightarrow} \\ 1 :: \langle 2, 3 \rangle &\quad \ddot{\rightarrow} \\ 1 :: 2 :: \langle 3 \rangle &\quad \ddot{\rightarrow} \\ 1 :: 2 :: 3 :: \langle \rangle &\end{aligned}$$

As yet another example,

$$\text{Define}(\text{Macro}, x = y, (z) x = y = z \ddot{\rightarrow} x = y \wedge y = z)$$

has the effect that e.g.

$$1 + 2 + 3 + 4 = 3 + 3 + 4 = 6 + 4 = 10$$

macroreduces to

$$1 + 2 + 3 + 4 = 3 + 3 + 4 \wedge 3 + 3 + 4 = 6 + 4 \wedge 6 + 4 = 10$$

achieving the same effect as the Mizar [12,16] .= operator using macroreduction.

In the macro definition above, x and y are meta-variables because they occur as arguments to the symbol being defined and z is a meta-variable because it is quantified by a meta-quantifier.

A particularly important macro reads

$$\text{Define}(\text{Macro}, (x), (x) \ddot{\rightarrow} x$$

which says that parentheses should be removed from parsetrees before proof checking.

6.2 Macrodefined definitions

The macro facility allows massive simplifications of the syntax. First of all, the definition

$$\text{Define}(\text{Macro}, x \doteq y, x \doteq y \ddot{\rightarrow} \text{Define}(\text{Macro}, x, x \ddot{\rightarrow} y))$$

allows to abbreviate

$$\text{Define}(\text{Macro}, h(x), h(x) \ddot{\rightarrow} x^2 - 1)$$

as

$$h(x) \doteq x^2 - 1$$

As another example,

$$x \doteq y \doteq \text{Define}(\text{Value}, x, y)$$

is the definition of $x \doteq y$.

6.3 Macros for metalogic

The following macro definitions define several constructs that were used to introduce rules, theories, lemmas, and proofs in Section 2.3:

$$\text{Rule } x: y \doteq \text{Define}(\text{Rule}, x, y)$$

$$\text{Theory } x: y \doteq \text{Define}(\text{Theory}, x, y)$$

$$x \text{ Lemma } y: z \doteq \text{Define}(\text{Lemma}, y, x \#- z)$$

$$\text{Proof of } x: y \doteq \text{Define}(\text{Proof}, x, y)$$

The construct $x \#- z$ expresses that the term z is a lemma of the theory x . For a complete list of meta-connectives see Section 8.5.

7 Programming language

Like most other proof systems, Logiweb includes a programming language. The programming language of Logiweb allows to express side conditions, proof tactics, complicated macros, complex rendering schemes, and many other things. Like e.g. Nuprl [4] and Isabelle [13,14], Logiweb uses a programming language that is based on lambda calculus. Logiweb uses a very simple, untyped version of lambda calculus. Simplicity was chosen to make it easy to reason about Logiweb and easy to implement it at the (minor) expense that programming has to start at a quite low level.

The chosen programming language is the programming language of Map Theory [2,7,15,17] and extends λ -calculus [3] with the constant **nil** and the construct **ifnil**(x, y, z). The **ifnil**(x, y, z) construct has the property that the range of the function $\lambda x. \mathbf{ifnil}(x, y, z)$ equals $\{y, z, \Omega\}$ (where $\Omega \doteq \mathbf{apply}(S, S)$, $S \doteq \lambda x. \mathbf{apply}(x, x)$). Hence, the programming language includes functions whose range have exactly three elements, which is impossible in pure λ -calculus [1]. This property allows to reason about programs using a “quartum non datur” rule [2,7,15,17] which essentially says that **ifnil**(x, y, z) can take no other values than y , z , and Ω . Furthermore, the presence of **ifnil** allows the programming language to have very simple, non-trivial models that are more semantic and less syntactic than those of pure λ -calculus.

7.1 Evaluation

The programming language of Logiweb provides four basic constructs: $\lambda x.y$, **apply**(x, y), **ifnil**(x, y, z), and **nil**. The definition $f(x) \doteq x^2 - 1$ and claim $f(10) = 100$ presupposes that x^y , $x - y$, $x = y$, and numerals are defined from these four constructs (c.f. [8]).

When a browser verifies a claim, it reduces the claim according to the

following reduction rules using normal order reduction [1]:

$$\begin{aligned} \mathbf{ifnil}(\mathbf{nil}, y, z) &\rightarrow y \\ \mathbf{ifnil}(\lambda u.v, y, z) &\rightarrow z \\ \mathbf{apply}(\mathbf{nil}, z) &\rightarrow \mathbf{nil} \\ \mathbf{apply}(\lambda x.y, z) &\rightarrow \langle y|x:=z \rangle \end{aligned}$$

Terms other than the four above are reduced by looking up their value aspect in the codex and using that. If a term has no value aspect, it reduces to **nil**.

A claim is accepted if it reduces to **nil** and is rejected if it reduces to a term of form $\lambda x.y$.

The Logiweb programming language is also used when verifying side conditions: When checking a proof of form e.g. $((x, y) (\neg\text{free}(x, y) \Vdash \exists x: y \in x)) \gg \exists u: \emptyset \in u$, a browser binds x and y to denote the parsetrees of u and \emptyset , respectively, and checks that the side condition $\neg\text{free}(x, y)$ reduces to **nil** according to whatever definitions there are of $\neg p$ and $\text{free}(p, q)$ in the codex.

As mentioned, the Logiweb programming language is also used in many other situations in the system.

7.2 Connection to Map Theory

Logiweb was originally developed to support Map Theory [2,7,15,17]. Map theory is the reduction system above extended with a quantifier and has the power of ZFC set theory.

Logiweb has been detached from Map Theory, however, so that Logiweb supports all theories equally well. One exception is that, since Logiweb uses the programming language embedded in Map Theory as user programming language, Map Theory is particularly well suited for reasoning *about* Logiweb, i.e. to serve as meta-theory for the system.

8 Verification

A major feature of any proof system is the ability of the system to verify the correctness of mathematics presented to it. Logiweb takes the approach that each author may make “claims” and it is then up to Logiweb to try to verify the claims. A page is correct if its claims are correct. A page that makes no claims is trivially correct. The following pages describe various kinds of claims and various facilities for expressing them.

8.1 Page symbols

As mentioned, a codex associates values to reference-symbol-aspect triples $\langle r, s, a \rangle$. Occasionally, there is a need to define aspects of a page rather than

aspects of a symbol. As an example, the claims of a page associate to the whole page.

To cope with that and to keep codices homogeneous at the same time, the convention is made that a symbol $\langle p, 0 \rangle$ whose id equals zero represents the page p . Symbols whose id equals zero will be referred to as “page symbols”. Logiweb forces all page symbols to have arity zero.

Logiweb provides the construct “Self” that macroreduces to the parsetree $\langle \langle r, 0 \rangle \rangle$ where r is the reference of the page that Self occurs on.

8.2 Simple claims

The following macro definition defines the construct for making simple claims:

$$[x] \doteq \text{Define}(\text{Test}, \text{Self}, x)$$

Hence, a claim like $[f(10) = 99]$ is shorthand for

$$\text{Define}(\text{Test}, \text{Self}, f(10) = 99)$$

For that reason, a claim made on a page enters the codex, and it is up to the browser to access and execute the claim in the codex when the user wants to verify a page.

8.3 Multiple definitions

If a page contains more than one definition of the same aspect of the same symbol, then the browser collects a list of all the definitions and then synthesizes a single definition from all the definitions. By default, a definition is synthesized from a list as follows: If all the definitions are identical, then one of them is used and otherwise the codex stores a value that indicates that the definition is in error. The claim aspect is treated differently, however, in that all the definitions are joined into a conjunction. For that reason it is possible to make more than one claim on a page.

8.4 Overall claims

In addition to simple claims like $[f(10) = 99]$, a page also makes an overall claim through a definition like

$$\text{Define}(\text{Claim}, \text{Self}, \mathcal{T})$$

If a page makes an overall claim using a definition like the one above, then the page is considered correct if **apply**(\mathcal{T} , Self) evaluates to **nil**. If \mathcal{T} implements e.g. Mizar [12,16], then the page is claimed to be Mizar-correct. Such a term \mathcal{T} is likely to be defined using a substantial number of definition, possibly scattered over many pages, and have to be expressed in the programming language of Logiweb. Including other systems like Mizar in Logiweb in this way has the benefit, among other, that each version of the system becomes frozen and correctness of a page will always be relative to one, particular version of

the system. Hence, each correctness claim will be constant even if the system in question is under development.

If a page has no overall claim (or the overall claim is in error), a browser should check the page using the default claim which says that all simple claims should evaluate to **nil**, all proofs should be correct, and no definitions should be marked as erroneous.

8.5 *Metalogic*

The default overall claim recognizes the following syntax for rules, theories, lemmas, and proofs:

$$\begin{aligned} \text{Rule} & ::= \text{Term} \vdash \text{Rule} \mid \text{Term} \Vdash \text{Rule} \mid (\text{Term}) \text{Rule} \mid \text{Term} \\ \text{Theory} & ::= \not\vdash \text{Term} \mid \text{Theory} \oplus \text{Theory} \mid \text{Rule} \\ \text{Lemma} & ::= \text{Theory} \Vdash \text{Term} \\ \text{Proof} & ::= \text{Proof} \triangleright \text{Proof} \mid \text{Proof} \gg \text{Term} \mid \text{Term} : \text{Proof} \mid \\ & \quad \text{Proof}; \text{Proof} \mid \text{Rule} \end{aligned}$$

Above, a Term denotes any parsetree that does not contain any of the meta-constructs ($x \vdash y$, $x \Vdash y$, and so one). $x \vdash y$ states that y is provable if x is provable. $x \Vdash y$ states that y is provable if x evaluates to “true” and, hence, allows to express side conditions. $(x) y$ states that y holds for all terms x . A term stated as a rule denotes an axiom. $\not\vdash x$ claims that x is non-provable in the theory it occurs in which allows to talk about the consistency of the theory. $x \oplus y$ denotes the theory that contains all rules of the theories x and y . A rule stated as a theory denotes a one-rule theory. $x \Vdash y$ denotes the lemma that y is a theorem of the theory x . $x \triangleright y$ denotes meta-modus-ponens applied to the conclusions of the proofs x and y . $x \gg y$ states that the term y follows from the proof x . $x : y$ locally introduces x as shorthand for the conclusion of the proof y . $x ; y$ has the same conclusion as y and allows y to use the conclusion of x as if it were an axiom. A rule stated as a proof has the rule itself as its conclusion and requires the rule to belong to the theory being used.

9 Bootstrapping

As mentioned, Logiweb offers complete notational freedom to its user. While this is very convenient, it complicates the problem of getting started. Logiweb offers great flexibility in defining new constructs from already known ones, but how can one introduce the first construct out of nothing? The present section deals with this bootstrapping problem.

Section 9.2 explains how all predefined constructs are introduced using a single “proclamation” construct, and then Section 9.3 reveals how to introduce this ultimate construct.

9.1 *Predefined symbols*

Symbols of form $\langle 0, i \rangle$ will be referred to as “predefined” symbols. Predefined symbols have a meaning which is fixed by the natural number i .

References to pages are always positive integers, and all references of all symbols in a parsetree refer to pages that are on store in the Logiweb system, so predefined symbols cannot occur in a parsetree. Predefined symbols can occur in the codex, however.

9.2 *Proclamations*

Suppose small-a denotes a parsetree for which the id of the root symbol equals the ASCII code for a small “a” (i.e. the id equals 97). Suppose small-b, small-c, and so on have similar properties. Further suppose $x \cdot y$ is a right associative construct of arity 2 and End is a construct of arity 0. A proclamation like

Proclaim(**apply**(x, y), small-a · small-p · small-p · small-l · small-y · End)

defines the construct **apply**(x, y) to denote functional application. When a Logiweb browser sees a definition like the one above, it notes in the codex that the given page defines the Value aspect of **apply** to be $\langle \langle 0, i \rangle \rangle$ where i is the id that identifies functional application.

The proclamation construct Proclaim(x, y) allows to attach arbitrary symbols to arbitrary, predefined concepts. As examples, **apply**(x, y), Value, Message, and Define(x, y, z) are defined by proclamations. The proclamation construct even allows to define new proclamation constructs.

9.3 *Base pages*

A page whose bibliography is empty will be referred to as a “base page”. If r is the reference of a base page then a browser should regard $\langle r, 1 \rangle$ as a proclamation construct.

Hence, the procedure for bootstrapping on Logiweb is as follows: Publish a base page. Use the proclamation construct of the base page to associate symbols to all the other predefined constructs. Then use these symbols to make definitions, tests, rules, theories, lemmas, proofs, and so on.

9.4 *System supplied aspects*

The user can define aspects of symbols, but a few aspects are supplied by the Logiweb browser:

Every page is ultimately stored as a “vector”, i.e. a list of bytes. When a browser loads the page with reference r , it stores the vector of the page as the Vector aspect of the page symbol. In other words, the browser associates $\langle r, \langle r, 0 \rangle$, “Vector”) to the vector of the page.

Every page contains a bibliography which is a list of references, and a parsetree; a browser stores them as the Bibliography and Parsetree aspects,

respectively, of the page symbol of the page.

If the page is a base page (i.e. if the bibliography of the page is empty), then set the Codify aspect of symbol number 1 to $\langle\langle 0, i \rangle\rangle$ where $\langle 0, i \rangle$ is the predefined symbol that identifies proclamation constructs.

Every page defines the arity of each symbol. For each symbol with non-zero arity, a browser stores the arity as the Arity aspect of the symbol.

When a browser loads a page, it also loads all transitively referenced pages if they are not already in the codex. However, the browser also constructs a subset of the codex which contains the transitively referenced pages only and stores this subset as the Cache aspect of the page symbol.

Finally, when a browser has loaded a page, it adds the part of the codex associated to the page to the subset above to obtain the “local codex” of the page (the part of the codex relevant to the given page). Then it stores the local codex as the value aspect of the page symbol. The Logiweb evaluator merely has access to the value aspect of symbols, but can access all of the codex relevant to the given page through this mechanism. It follows that the Self macro always expands to something whose value is the local codex.

A The Logiweb protocol layer 0

A.1 Platform

At the hardware level, Logiweb is supported by a number of host machines that are interconnected by a number of host networks. As an example, if Logiweb runs on a number of pc’s that are interconnected by the Internet, then the pc’s are the host machines and the Internet is the host network.

Computers and networks that host Logiweb need not be dedicated to Logiweb alone. Rather, computers and networks that host Logiweb will typically host other systems as well.

At the software level, each implementation of the Logiweb server or Logiweb browser is supported by a host operating system (e.g. Linux), a host programming language (the implementation language) and a number of host protocols (e.g. TCP/IP).

A.2 Layers

The Logiweb protocol has three layers. Layer 0 is concerned with the storage, retrieval, and transmission of bytes. Layer 0 defines the behavior of Logiweb servers. Layer 1 is concerned with the unpacking of pages into codices, and the checking and rendering of pages. Layer 1 defines the minimal functionality of Logiweb browsers. Layers above layer 1 are user defined and outside the scope of the Logiweb protocol. This chapter describes layer 0 of the protocol.

A.3 Bit and byte numbering conventions

Within a byte, we refer to the bit that has weight 2^i as bit number i . Hence, the least and most significant bits are bits number 0 and 7, respectively. In TCP/IP, bit number 0 is transmitted first and bit 7 is transmitted last.

Within a byte vector, we refer to the first byte as byte number 0. In TCP/IP, byte number 0 is the first byte transmitted. When writing byte vectors in this paper, byte number 0 is written leftmost.

When writing bytes as eight bits in this paper, bit number 0 is written leftmost, which is opposite to normal practice. This is done to avoid problems with the numbering of bits within sequences of bytes.

A.4 The format of pages

Bytes have the following format:

bit ::= 0 | 1

byte ::= bit⁸

The value of a byte $b = b_0b_1 \cdots b_7$ is

$$v_{\text{byte}}(b) = \sum_{i=0}^7 b_i \cdot 2^i$$

As an example,

$$v_{\text{byte}}(0010\ 0000) = 4$$

Cardinals (i.e. natural numbers) are represented as byte vectors as follows:

middle-byte ::= bit⁷ 1

end-byte ::= bit⁷ 0

cardinal ::= middle-byte* end-byte

The value of a cardinal $c = b_0b_1 \cdots b_n$ where b_1, \dots, b_n are bytes is

$$v_{\text{cardinal}}(c) = \sum_{i=0}^n (v_{\text{byte}}(b_i) \bmod 128) \cdot 128^i$$

As an example,

$$v_{\text{cardinal}}(0100\ 0001\ 1000\ 0000) = 2 + 128 = 130$$

Time stamps are represented as a mantissa and an exponent:

mantissa ::= cardinal

exponent ::= cardinal

timestamp ::= mantissa exponent

The value of a timestamp $t = m e$ where m and e are cardinals is

$$v_{\text{timestamp}}(t) = v_{\text{cardinal}}(m) \cdot 10^{-v_{\text{cardinal}}(e)}$$

As far as Layer 0 is concerned, the syntax of a page is thus:

```

version ::= 1000 0000
key     ::= byte20
body    ::= byte*
reference ::= version key timestamp
contents ::= timestamp body
page    ::= version key timestamp body

```

We shall refer to the concatenation of the version, key, and timestamp as the “reference” of a page and to the concatenation of the timestamp and body as the “contents” of a page:

$$\text{page} ::= \overbrace{\text{version key timestamp}}^{\text{reference}} \text{body}$$

$$\text{page} ::= \text{version key } \underbrace{\text{timestamp body}}_{\text{contents}}$$

The key of a page must be the RIPEMD-160 key of the contents of the page.

Note that a “version” is a byte whose value is 1. Later versions, if any, should be assigned other positive cardinals. In some situations described later, a single null byte is used to denote a null reference. For that reason, null bytes cannot be used as a version number. The definition of predefined symbols in Section 9.1 also relies on version numbers being non-null so that references to pages are forced to be non-null.

A.5 *Server and file names*

Every server must have at least one name (e.g. “`yoa.dk:65535`”, which is expected to be the address of the first Logiweb server, c.f. <http://yoa.dk/>) and every page on each server must also have at least one name (e.g. “`aux/base`”). The syntax of names is

```

length ::= cardinal
string  ::= byte*
name    ::= length string

```

In a name, the value of the length field must equal the length of the string. Logiweb does not interpret names. It just passes them around or passes them to network drivers or file system interfaces.

A.6 *Server states*

A Logiweb server has to maintain a “state” and must be prepared to answer questions about its state.

A state is a binary, labeled tree with edge-labels in $\{0, 1\}$. For every node N of the state, the edge labels of the path from the root to N form a sequence of bits which will be referred to as the “address” of N .

A node is said to be a “page” node if its address is a valid reference. Page nodes are required to be leaf nodes.

A node is said to be a “branch” node if it is not a leaf node. Hence, the page and branch nodes form disjoint sets. Branch nodes have two downward edges.

Nodes that are neither page nor branch nodes are called pointer nodes. Hence, pointer nodes are leaf nodes that are not page nodes. The page, branch, and pointer nodes form a class division of all nodes of the tree.

Each node of the tree is decorated by a “sibling list” and a “name list”. Both are lists of names where the notion of a name was introduced in Section A.5. The name list of branch nodes must be empty. The name list of page nodes must be non-empty.

If A is a sequence of bits, if two servers both have a node with address A , and if the two nodes have the same kind (page, branch, or pointer), then the two servers are said to be A -siblings. The sibling list of a node with address A on a server must be a list of names of servers that the present server thinks are A -siblings of the server. Any server must check periodically that the servers it thinks are A -siblings really are A -siblings. When a server changes or deletes a node with address A , it must inform its A -siblings about the change. When a server creates a node, it must try to locate a reasonable amount of A -siblings. As an ideal, all A -siblings should be connected transitively through sibling pointers.

The root of a tree must be a branch node. The sibling list of the root must be edited manually by the system manager whereas all other sibling lists are maintained automatically. The siblings of the root node of a server are called the “friends” of the server.

When a new server is started, its friends must be chosen manually to ensure good connection to other servers, and maintainers of other servers should be persuaded to include the new server in their friend lists.

The name list of a page node is a list of file names where each file stores a copy of the given page. Individual users of a server may submit and delete the same page, and the name list keeps track of the copies currently on store. When the last copy of a page is deleted, the server must delete the page node and inform siblings about the loss.

The name list of a pointer node with address A is a list of server names that the present server thinks have branch nodes with address A . Any server must check periodically that the servers it thinks have such branch nodes really do store such nodes.

A.7 Layer 0 protocol

Two servers or a server and a browser may communicate with each other via TCP or UDP or some other protocol. When they do so, they send messages to each other. If they use a connection based protocol like TCP, they communicate using a continuous stream of messages that are put back-to-back on the stream. If they communicate using a datagram protocol like UDP, they may send each messages in a separate package, or they may pack several messages into a single package, but they cannot break a message into several packages.

In the following, the numbers 0, 1, . . . , 255 denote bytes.

prefix	::= cardinal
kind	::= cardinal
names	::= cardinal
siblings	::= cardinal
index	::= cardinal
password	::= cardinal
length	::= cardinal
nop	::= 0
sorry	::= 1
ping	::= 2 reference
ping2	::= 3 reference timestamp
node	::= 4 reference prefix
branch	::= 5 reference prefix siblings
page	::= 6 reference prefix siblings names length
pointer	::= 7 reference prefix siblings names name
name	::= 8 reference prefix index
name2	::= 9 reference prefix index name
sibling	::= 10 reference prefix index
sibling2	::= 11 reference prefix index name
retrieve	::= 12 reference password index length
retrieve2	::= 13 reference password index length byte*
translate	::= 14 name
translate2	::= 15 name reference
message	::= nop sorry ping ping2 node branch page pointer name name2 sibling sibling2 retrieve retrieve2 translate translate2

When a server receives a “nop” message, it should do nothing. Such nop messages are useful for padding.

When a server is too pressed to deliver its service, it sends a “sorry” message and then either slows down or simply breaks the connection.

When a server receives a “ping” message it adds a time stamp that in-

icates the current server time, changes “ping” to “ping2”, and returns the message to sender.

The node, name, and sibling messages all query information about a particular node of the state tree of the server. Each of these messages contain a reference and a prefix where the prefix is a cardinal. Let P denote the value of the cardinal. Let A' denote the first P bits of the reference (or the whole reference if P is greater than or equal to the number of bits of the reference). Let A be the longest prefix of A' for which the server has a node with address A (in particular, $A = A'$ if the server has a node with address A').

When a server receives a node message it computes A as above and returns a branch, page, or pointer message if its node at address A is a branch, page, or pointer node, respectively. In each case the server indicates the length of the sibling list of the node. For page and pointer nodes it also indicates the length of the name list of the node. For page nodes, it indicates the length of the stored page measured in bytes, and for pointer nodes it provides a pseudorandom entry from its name list.

When a server receives a name or sibling message, it computes A as above and computes the value I of the given index. Then it returns a name2 or sibling2 message containing the I 'th element of the name or sibling list, respectively, of the node with address A .

When a server receives a retrieve message, it may look up the given page and return “length” bytes of the page starting at position “index”. It may send the requested bytes in a single retrieve2 message or may break it into several retrieve2 messages.

A suspicious server, however, may return a few bytes of the page and a “password”, where a password is an integer. The requester then has to ask for the page once more, using the given password. This feature is only relevant in connection with datagram protocols where the authenticity of the sender is difficult to verify and where malicious users could otherwise use Logiweb servers for denial-of-service attacks.

If a requester has no password to offer, the password field of a retrieve message should be set to zero.

If the length plus the index of a retrieve message is greater than the length of the page, the server shall unconditionally ignore the retrieve message.

Using the messages presented so far, one may query all information about the state of a server. When the state of a server changes, the server should send unsolicited branch, page, and pointer messages to relevant siblings.

Two additional messages allow to translate local file names to references which is mainly of interest for authors who publish pages on the server. When a server receives a translate message, it looks up the given file and returns a translate2 message with the associated reference. Whenever a user creates or deletes a page in the servers file system, the user should send the server a translate message to make the server notice the change.

B The Logiweb protocol layer 1

B.1 Overview

To Logiweb servers, a page consists of a reference and a body where a body is an uninterpreted sequence of bytes. Logiweb browsers and Layer 1 of the protocol are concerned with the structure of page bodies.

The user of a Logiweb browser may direct the browser to “load” a page. The input to this loading process is the reference of the page to load. The output is new entry for the codex. Loading is recursive in the sense that loading a page may request the browser to load transitively referenced pages as well.

The following sections describe some data structures and then describes the process of loading pages.

The description merely describes what happens in principle. A straightforward implementation may lead to poor execution speed and unreasonable memory requirements. Implementors are free to optimize representations as long as optimizations are invisible to the user.

B.2 Binary trees

A Logiweb binary tree is a structures with the following syntax:

$$\text{tree} ::= \mathsf{T} \mid (\text{tree} :: \text{tree})$$

In principle, a codex is such a tree. In the language of Section 7.1, trees are represented thus:

$$\begin{aligned} \mathsf{T} &\doteq \mathbf{nil} \\ x :: y &\doteq \lambda z. \mathbf{ifnil}(z, x, y) \end{aligned}$$

As in Section 6, $\langle x_1, \dots, x_n \rangle$ means $x_1 :: \dots :: x_n :: \mathsf{T}$.

B.3 Cardinals

We use T and $\mathsf{T} :: \mathsf{T}$ to represent the bits 0 and 1, respectively. In principle, cardinals (i.e. finite cardinals/natural numbers) are represented as lists $\langle b_0, \dots, b_n \rangle$ of bits where b_0 is the least significant bit and the most significant bit b_n is required to represent 1. The empty list represents zero. As an example of use, $\langle \mathsf{T}, \mathsf{T}, \mathsf{T} :: \mathsf{T} \rangle$ represents the cardinal “four”.

B.4 Arrays

We shall say that a list of pairs

$$\langle k_1 :: v_1, \dots, k_n :: v_n \rangle$$

is an “array” if k_1, \dots, k_n is a strictly decreasing sequence of cardinals and v_1, \dots, v_n are trees all of which differ from T . For an array A and a cardinal k let $A[k]$ denote v_i if $k = k_i$ for some i and let $A[k] = \mathsf{T}$ if $k \neq k_i$ for all i .

We shall say that a function f from cardinals to trees is “finite” if $\{n \mid f(n) \neq \top\}$ is finite. For all finite functions f there exists one and only one array A such that $f(n) = A[n]$.

B.5 Multi-dimensional arrays

We shall refer to any tree as a zero-dimensional array. Furthermore, we shall refer to an array

$$\langle k_1 :: v_1, \dots, k_n :: v_n \rangle$$

for which all v_i are d -dimensional arrays as a $(d+1)$ -dimensional array.

If A is a d -dimensional array and i_1, \dots, i_d are cardinals, then $A[i_1] \cdots [i_d]$ is a tree.

B.6 The format of codices

A codex is a five-dimensional array. For a codex C , the value of

$$C[p_r][s_r][s_i][a_r][a_i]$$

denotes the $\langle a_r, a_i \rangle$ -aspect of symbol $\langle s_r, s_i \rangle$ as defined on page $\langle p_r \rangle$. Such an aspect may be \top (indicating that no aspect is defined), or it may be a pair $m :: v$ where m and v are the “mode” and “value” of the aspect, respectively. If $m = 0$ then the aspect is defined and v is the value of the aspect. If $m = 1$ then the aspect is noted to be in error. Other values of m are used during codification.

B.7 Overview of loading

When a browser loads a page, it starts out with a page reference p_r and a codex C and then does as follows:

codex check The browser looks in its codex and skips the rest of the loading if the page is already there.

retrieval The browser uses Layer 0 of the protocol to locate and retrieve the page.

bibliography parsing The browser parses the body of the page to get the bibliography.

first page loading The browser loads the first page mentioned in the bibliography (if any).

unpacking The browser parses the rest of the page to get an arity table and a parsetree in Polish prefix notation.

transitive loading The browser loads all transitively referenced pages.

parsing The browser constructs the parsetree of the page.

codification The browser translates the parsetree into a codex entry C' and performs $C[p_r] := C'$.

The following sections describe the loading process.

B.8 Retrieval

The browser locates the page as follows: Set the “current server” to be the local server. Then iterate the following: Send a node message with the requested reference and the length of the requested reference to the current server. If the response is a page message, the page is on the current server. If the response is a pointer message with a null server name, give up. Otherwise, set the current server to the returned server name and iterate. If and when the page is located, retrieve it using retrieve messages.

B.9 Bibliography parsing

The browser parses the body of the page according to the syntax

$$\text{body} ::= \text{reference}^*0\text{remainder}$$

where the zero denotes a null byte.

When parsing, if the parser reads past the end of the body, the body is padded with zero bytes. Hence, when publishing a page, it is legal to cut away trailing zero bytes.

B.10 Unpacking

The browser looks up the “Unpack” aspect of the page symbol of the first page in its bibliography. If that aspect has form $0 :: u$ then the result of the unpacking is **apply**($u' R$) where R is the remainder from Section B.9 expressed as a tuple of bytes where bytes are cardinals. This feature allows to override the default encoding of pages.

If the Unpack aspect does not have form $0 :: u$ then the browser parses the remainder according to the following format:

$$\begin{aligned} \text{id} & ::= \text{non-zero-cardinal} \\ \text{arity} & ::= \text{cardinal} \\ \text{arities} & ::= \{\text{id arity}\}^* 0 \\ \text{parsetree} & ::= \text{cardinal}^* \\ \text{remainder} & ::= \text{arities parsetree} \end{aligned}$$

When parsing the parsetree, the parser stops when there are no non-null bytes left.

The result of the parsing is a pair $A :: P$ where A is an array that maps id's to arities as specified and P is a “linear parsetree” represented as a list of cardinals (essentially a parsetree in Polish prefix notation). Each byte in the parsetree of the byte vector contributes up to 7 bits to P since one bit of

each byte is used to tell where each cardinal ends and since leading zero bits are suppressed.

If more than one arity is assigned to the same id, the last assignment takes precedence. There is no difference between assigning an arity of zero and assigning no arity.

B.11 Transitive loading

When loading a page, the reference of the page is regarded as entry number zero in the bibliography of the page. In this perspective, every bibliography has at least one entry, and a page is a base page (c.f. Section 9.3) if its bibliography merely has one entry.

For all page references p let $|p|$ denote the number of entries in the bibliography of p and let $\lceil p \rceil$ denote the smallest power of two which is greater than or equal to $|p|$. Let $p[i]$ denote the i 'th entry in the bibliography of p .

The linear parsetree P from Section B.10 is a list of cardinals. Such a cardinal c on a page with reference p represents a symbol $s(c, p)$ of a transitively referenced page according to the following scheme:

$$s(c, p) = \begin{cases} s(c \operatorname{div} \lceil p \rceil, p[c \operatorname{mod} \lceil p \rceil]) & \text{if } 0 < c \operatorname{mod} \lceil p \rceil < |p| \\ p :: (c \operatorname{mod} \lceil p \rceil) & \text{otherwise} \end{cases}$$

In this way, all symbols on all transitively referenced pages are represented by cardinals in a way that is compact and fairly easy to decode using bit shift operations.

When the browser does transitive loading, it translates all cardinals in the linear parsetree P from Section B.10 to symbols and loads the home pages of all the symbols. During this process the browser may need to find the bibliographies of pages that it does not need to load. Such bibliographies are stored in the codex in the same way as bibliographies of loaded pages, but the browser does not otherwise load such auxiliary pages.

B.12 Parsing

The browser stores the arities from Section B.10 in the codex as mentioned in Section 9.4. Then it converts the linear parsetree from Section B.10 from Polish prefix notation to a tree using the arities in the codex and at the same time converts each cardinal c in the linear parsetree to the symbol $s(c, p)$.

B.13 Codification

The browser enters the system supplied aspects mentioned in Section 9.4 into the codex attaching mode 0 (c.f. Section B.6) to each of them. Then the browser scans the parsetree for proclamations and enters them into the codex attaching mode 0 to each of them.

Then the browser scans the parsetree from Section B.12 for definitions. The parsetree is macroreduced during this scan. Macro definitions from the page itself have not yet been recognized, so they have no effect on this scan. Definitions that try to change definitions already in the codex are ignored. Other definitions are collected as codex entries of form $\langle 2, v_1, \dots, v_n \rangle$ where 2 is a temporary mode of the definition and v_1, \dots, v_n are definitions (c.f. Section 8.3 concerning multiple definitions).

Proclamations are treated before definitions to ensure that a construct that is proclaimed to be e.g. a definition construct can be used for making definitions on the page where it is proclaimed.

Then the browser synthesizes definitions as follows: It scans the codex for definitions with mode 2. When it encounters a definition of mode 2, it changes the mode to 3 and starts synthesizing the definition. By default, it synthesizes a definition $\langle 2, v_1, \dots, v_n \rangle$ by macroreducing v_1, \dots, v_n , comparing the results, and using v_1 as the definition of the construct if all the v_1, \dots, v_n are identical except for naming of bound variables. If v_1, \dots, v_n contradict each other, the browser changes the mode of the definition to 1 to indicate error.

As an exception, the Test aspect is synthesized by forming the conjunction of the individual definitions.

During macroreduction, if the browser has to macroreduce a symbol whose macro aspect has mode 2, then the browser synthesizes that macro aspect first. If the browser has to macroreduce a symbol whose macro aspect has mode 1 or 3 (i.e. is in error or is being synthesized) then the definition the symbol occurs in gets mode 1 to indicate error.

References

- [1] Barendregt, H., “The Lambda Calculus, Its Syntax and Semantics,” Studies in Logic and The Foundation of Mathematics **103**, North-Holland, 1984.
- [2] Berline, C. and K. Grue, *A κ -denotational semantics for Map Theory in ZFC+SI*, Theoretical Computer Science **179** (1997), pp. 137–202.
- [3] Church, A., “The Calculi of Lambda-Conversion,” Princeton University Press, 1941.
- [4] Constable, R. L., S. Allen, H. Bromly, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. Sasaki and S. Smith, “Implementing Mathematics with the Nuprl Proof Development System,” Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
- [5] de Bruijn, N. G., *A survey of the project AUTOMATH*, in: J. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, 1980 pp. 579–606.
- [6] Dobbertin, H., A. Bosselaers and B. Preneel, *RIPEDM-160: A strengthened*

- version of RIPEMD*, in: *Fast Software Encryption*, 1996, pp. 71–82, <http://citeseer.nj.nec.com/dobbertin96ripemd.html>.
- [7] Grue, K., *Map theory*, Theoretical Computer Science **102** (1992), pp. 1–133.
- [8] Grue, K., “Mathematics and Computation,” (lecture notes) **1–3**, DIKU, 2001, 7. edition, <http://www.diku.dk/~grue/papers/mac0102/>.
- [9] Grue, K., *Map theory with classical maps*, Technical Report 02/21, DIKU (2002), <http://www.diku.dk/publikationer/tekniske.rapporter/2002/>.
- [10] Kohlhase, M., *OMDoc: An open markup format for mathematical documents (version 1.1)* (2003), <http://www.mathweb.org/omdoc.ps>.
- [11] Miner, R. and J. Schaeffer, *A gentle introduction to MathML* (2001), <http://www.dessci.com/en/support/tutorials/mathml/default.htm>.
- [12] Muzalewski, M., “An Outline of PC Mizar,” Foundation of Logic, Mathematics and Informatics, Mizar User Group, Brussels (1993).
- [13] Paulson, L. C., *Introduction to isabelle*, Technical report, University of Cambridge, Computer Laboratory (1998).
- [14] Paulson, L. C., *The isabelle reference manual*, Technical report, University of Cambridge, Computer Laboratory (1998).
- [15] Skalberg, S. C., “An Interactive Proof System for Map Theory,” Ph.D. thesis, University of Copenhagen (2002), <http://www.mangust.dk/skalberg/phd/>.
- [16] Trybulec, A. and H. Blair, *Computer assisted reasoning with MIZAR*, in: A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence* (1985), pp. 26–28, <http://www.mizar.org/>.
- [17] Vallée, T., “*Map Theory*” *et antifondation*, Electronic Notes in Theoretical Computer Science **79** (2003), pp. 1–258.