The implementation of Logiweb

Klaus Grue
Dept.Comp.Sci. University of Copenhagen
grue@diku.dk

Abstract

This paper describes the implementation of the 'Logiweb' system with emphasis on measures taken to support classical reasoning about programs.

Logiweb is a system for authoring, storing, distributing, indexing, checking, and rendering of 'Logiweb pages'. Logiweb pages may contain mathematical definitions, conjectures, lemmas, proofs, disproofs, theories, journal papers, computer programs, and proof checkers.

Reading Logiweb pages merely requires access to the World Wide Web. Two example pages are available on `http://yoa.dk/`. Writing, checking, and publishing Logiweb pages requires Logiweb to be downloaded and installed.

Logiweb comes with a hierarchy of features: Lemmas and proofs are stated in a theory named 'Map Theory', Map Theory is implemented on top of a calculus named 'Logiweb sequent calculus', and Logiweb sequent calculus is implemented on top of the 'Logiweb reduction system' (a version of λ-calculus). The Logiweb reduction system is implemented in the Logiweb core software which is currently implemented in Common Lisp.

The levels above the Logiweb core software are defined on Logiweb pages, allowing users to use the features as they are or to define and publish new ones on new Logiweb pages. As an example, a user may want to use ZFC in place of Map Theory, in which case the easiest approach is to publish a Logiweb page that defines ZFC in Logiweb sequent calculus and proceed from there.

The 'base' page on `http://yoa.dk/`, which is 180 pages long when printed out, was checked in 40 seconds. This is non-trivial to achieve for a proof checker implemented in lamdba calculus.

The Logiweb sequent calculus is defined on the base page mentioned above. A user who wants to define e.g. ZFC set theory on top of that may publish a new page, call it 'zfc', and let the 'zfc' page reference the 'base' page. That makes all definitions on the 'base' page available to the 'zfc' page. After that, another user may state and prove lemmas about e.g. real numbers on a third page, call it 'real', which references the 'zfc' page. When the proofs on the 'real' page are checked, Logiweb will arrange that the 'zfc' and 'base' pages are available in a predigested form suitable for proof checking.

Seen from the point of view of proof checking and publication, the World Wide Web has the drawback that once submitted pages can be modified after submission. In the example above, modification of the 'base' page could ruin the correctness of the 'real' page.

To avoid problems with pages being modified, Logiweb implements its own referencing system which forces immutability upon once submitted pages. Once a Logiweb page is submitted, it cannot be changed, just like papers cannot change after publication.

When a Logiweb page is submitted, a unique Logiweb 'reference' is computed from its contents. The Logiweb system allows to look up a Logiweb page given its reference.

Once a Logiweb page is submitted, it may be moved and duplicated such that its http url may change and such that a page may be available many places in the world under different urls, but the Logiweb reference remains constant. One of the tasks of Logiweb is to keep track of the relation between the fixed references and the associated, fluctuating set of http urls.

1 Introduction

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

Logiweb is accumulative and provides a medium for archived mathematics. In contrast, the World Wide Web, which supports mathematics through MathML and OMDoc [Koh03, MS01], is a medium suited for information in flux.

Like the Internet and the WWW, Logiweb is a robust, 'anarchistic' system that runs without any central authority; it has been designed in the hope that such a system is the missing piece of software for widespread usage of automated reasoning.

Currently, Logiweb is used as it is, but it also has the potential to run silently and transparently underneath other systems like Mizar [Muz93, TB85]. Support for other systems requires substantial effort, but the hooks for doing so in many different ways are available in Logiweb.

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 [BG97, Gru92, Ska02, Val03] 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. However, Logiweb has been designed such that it supports all axiomatic theories equally well so the ability to support Map Theory should be seen as a widening rather than a narrowing of the scope.

Logiweb puts no restrictions on what logic is used in the sense that it can support any theory for which one can program a mechanical proof checker. The ease with which Logiweb supports highly distinct theories like ZFC and Map Theory indicates that use of arbitrary logic is not only possible but also feasible. Logiweb supports classical as well as intuitionistic logic, it supports theories built on first order predicate calculus as well as other brands of theories, and it supports theories (such as Map Theory) which admits general recursive definitions.

The absence of restrictions on the choice of logic of course makes it impossible to

supply a code-from-theorems extraction facility like `term_of` of Nuprl [CAB+86], but functions for manipulation of theorems and proofs of individual theories are expressible in the programming language of Logiweb.

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 [Gru01, Gru02] have been developed 1992-2002 to test the system.

Reference [Gru01] 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. The associated course has been given ten times with a total of more than a thousand students. The course has been a success and runs as the first course on the computer science curriculum at DIKU in parallel with a course on ML.

Reference [Gru02] 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, [Gru02] has been ported by hand to Isabelle [Pau98a, Pau98b, Ska02].

At the time of writing, Logiweb is used on a graduate course in logic (c.f. `http://www.diku.dk/~grue/logiweb/20050502/home/index.html`) and Logiweb is being adapted according to user requests. After that, it is the intension to run first [Gru02] and then [Gru01] through the system. Running those two books through Logiweb requires adaption of the books to the current syntax of the Logiweb compiler plus programming of a number of proof tactics that are described but not formally defined in the books. Running [Gru02] through Logiweb will also allow a comparison with Isabelle.

Map Theory essentially is the Logiweb programming language extended with a quantifier. As a long term goal, this makes it interesting to use Map Theory to reason *about* Logiweb, possibly leading to a situation where one can solve the academic exercise to let a proof checker prove its own correctness. A more immediate application is to use Map Theory to reason about code fragments expressed in the Logiweb programming language as is done in [Gru01].

1.1 Overview of the paper

Logiweb is a simple system with a simple programming language, a simple macro expansion facility, a simple proof checking facility, a simple protocol for exchange of documents, a simple format for storing Logiweb pages and so on. While each feature is simple in itself, the sum of features may make Logiweb look complex at first sight. For a comprehensive introduction to Logiweb, consult Logiweb itself at `http://yoa.dk/` and read the 'base' page.

The present paper gives an overview of the system from an implementation perspective in Section 1.2 and from a user perspective in Section 2. Then Section 3 describes how Logiweb pursues its goal to allow classical reasoning about programs without sacrificing generality and efficiency of computation. Section 4 describes the data structures used for representing terms, lemmas, proofs, pages, and so on. Section 5 describes the proof checking algorithm and Section 6 summarizes.

1.2 System overview

A user may use the World Wide Web as shown in Figure 1. In the figure, the user may use the text editor to construct an html page and store it in the file system within reach of the http server. Then the user (or another user) may use the html browser to request the html page from the http server which in turn retrieves the html page from the file system.

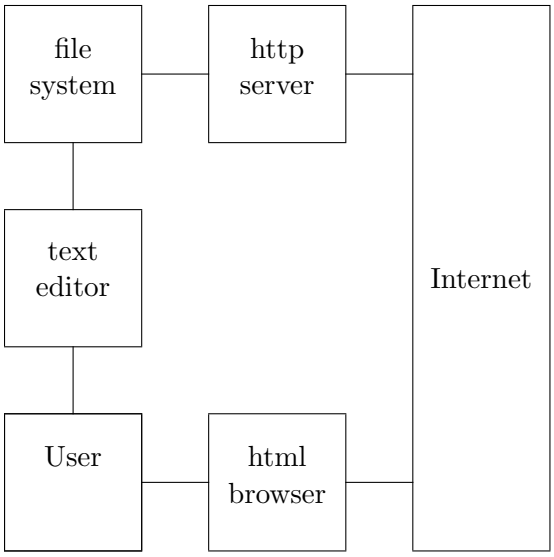

Figure 1: World Wide Web

Figure 2 shows how a user may use Logiweb. To write a Logiweb page, the user prepares a source text and invokes the Logiweb compiler on it. This is similar to running TEX on a TEX source [Knu83]. Actually, much of a Logiweb source consists of TEX source code.

When and if the compiler succeeds in interpreting the source, it translates it to a compressed format, checks its mathematical correctness, and stores it back in the file system in the format of a Logiweb page within reach of the http server. The compiler also renders the page in PDF so that users without a genuine Logiweb browser can view it. After that, any user that knows the url of the page can retrieve it using an html browser.

When the compiler succeeds in translating a Logiweb page, it also computes the Logiweb reference of the page and notifies the Logiweb server (c.f. Figure 2). The Logiweb server keeps track of the relationship between http urls and Logiweb references and makes the relationship available via the Internet using the Logiweb protocol. The Logiweb protocol allows Logiweb servers to cooperate on indexing pages such that each server merely has to keep track of local pages plus some information about which other Logiweb servers to refer non-local requests to.

A Logiweb reference contains a RIPEMD-160 [DBP96] hash key and a time stamp. The RIPEMD-160 hash key is computed on basis of the bytes of the associated page. As long as RIPEMD-160 stands up against collision attacks, not even a malicious user can get away modifying as much as a single byte of a Logiweb page without getting

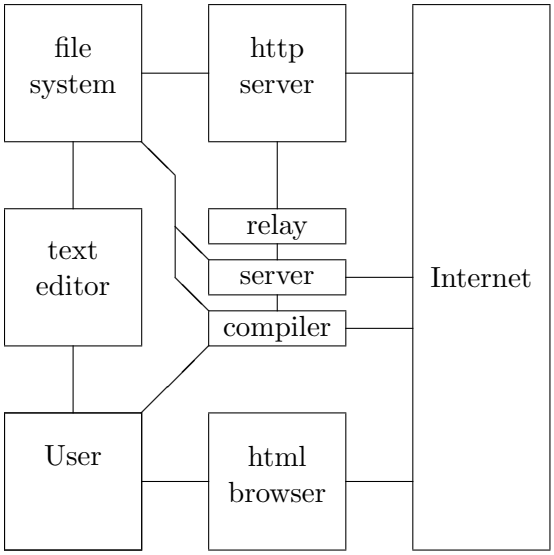

Figure 2: Logiweb

caught by a RIPEMD-160 check.

When the compiler translates a Logiweb page that references other Logiweb pages (which is the normal case), it uses the Logiweb server to locate the references and then transitively loads the referenced pages so that all definitions on transitively referenced pages are available.

When referencing a Logiweb page from the World Wide Web, one may construct an http url from the Logiweb reference by expressing the reference in hexadecimal and prepending it with the url of a Logiweb relay (c.f. Figure 2). A Logiweb relay is a CGI-program which, given a reference, contacts the nearest Logiweb server, translates the reference to an ordinary url, and returns an html indirection to that url. This instructs the html browser of the user to fetch the associated page. The net experience for the user is that clicking a Logiweb reference in an html document makes the html browser navigate to the referenced Logiweb page.

Referencing from Logiweb pages to html pages is trivial but not necessarily advisable since the immutability of Logiweb pages makes it impossible to repair broken links.

In addition to the Logiweb server, compiler, and relay mentioned above, the current implementation of Logiweb includes an 'lgwping' program which allows to ping a Logiweb server to see if it is responding.

For more details on Logiweb see `http://yoa.dk/` or [Gru04].

2 A Logiweb tutorial

2.1 Hello world

To give an overview of the system from a user perspective, we now follow the first steps of a new user. The steps are close to the steps actually followed by the current users (c.f. `http://www.diku.dk/~grue/logiweb/20050502/home/index.html`).

Previous versions of Logiweb offered a WYSIWYG authoring tool, but that has been abandoned until further and replaced by a lean and mean compiler that offers high speed and reasonably intelligible error messages, but no help beyond that. In other words, our new user is in a situation that resembles the situation of the first time user of a new programming language.

So a reasonable way to get started is to copy the source text of a "hello world" Logiweb page and try to compile that. The source of a "hello world" page is available at `http://www.diku.dk/~grue/logiweb/20050502/home/grue/hello-world/fixed/source/source.pyk`. The essential lines read

```
\begin{document}
"[ math pyk define hello world as "hello world" end define end math ]"
\end{document}
```

which requires quite a lot of explanation to make sense. Instead of looking for an explanation, our new user stores the source text in the file "page.pyk" and runs the compiler by issuing a command like the following:

```
> pyk pyk=page url=http://my.domain/my/directory level=all
```

After that, our user starts an html browser and looks up `http://my.domain/my/directory/hello-world/fixed`, then clicks "body", and then clicks "PDF" to see the following:

$$[\text{hello world} \stackrel{\text{pyk}}{=} \text{“hello world”}]$$

2.2 A minor update

Our new user, encouraged by seeing output from the system, modifies the source:

```
\begin{document}
The definition "[ math pyk define hello world as "hello world" end
define end math ]" defines the name of {\em my} page.
\end{document}
```

Then the user reruns the compiler and asks the html browser to reload the page to get

$$\text{The definition } [\text{hello world} \stackrel{\text{pyk}}{=} \text{“hello world”}] \text{ defines the name of } \textit{my} \text{ page.}$$

A key feature of Logiweb is that pages are immutable, so it may seem peculiar how easily the user changed `http://my.domain/my/directory/hello-world/fixed` above. To Logiweb, however, the two pages have different Logiweb references and the first "hello world" page was immutable as long as it existed. Immutability means that, given a Logiweb reference, one can locate the associate page (if it exists anymore) and, furthermore, one can check whether or not anybody has tampered with the page.

Our user invoked the compiler with a "level=all" argument. That indicates that the backend of the compiler should render not only the page itself but also a lot of additional material. A "level=submit" is equivalent to "level=all", but in addition requests the

compiler to notify the nearest Logiweb server about the submission and to store the page as `http://my.domain/my/directory/hello-world/TIME` where TIME is the date and time of submission. This is useful for versions of a page that are expected to exist for more than a debug round trip.

2.3 Guarding against haphazardness

Now our hopeful user is ready for doing some proof checking. However, suppose the source text of the "hello world" page contains something like

```
BIBLIOGRAPHY
base: "http://yoa.dk/logiweb/page/base/fixed/vector/page.lgw"
```

These lines tell the compiler that the "hello world" page references whatever Logiweb page happens to be at that particular URL at the moment the "hello world" page is translated. The `.lgw` file is the real Logiweb page in a standardized, binary format. If the referenced page is overwritten, and no copies of the page exists anymore, then the "hello world" page will be ruined. So to guard against this, the user issues the following command:

```
> pyk lgw=http://yoa.dk/logiweb/page/base/fixed/vector/page.lgw \
> url=http://my.domain/my/directory level=submit
```

That makes the compiler make a local copy of the given Logiweb page. The local copy will have exactly the same reference as the original, and the local copy ensures that the Logiweb page will remain in existence even if the original instance of the page ceases to exist. Then the user may look up the raw Logiweb reference at `http://yoa.dk/logiweb/page/base/fixed/reference/kana.html` and insert that in the bibliography:

```
BIBLIOGRAPHY
base:nani
nuse siti sete tiku kata  sana susu siku kitu naku
kake sisu suni nusa tini  tesa kika sutu siku neke
saku kesa keke seke kine  suki sise nasa natu
```

This ensures that the "hello world" page will reference the same Logiweb page each time the user re-translates the "hello world" page.

2.4 Defining a theory

Having guarded against the haphazardness of the external world, our user may write

```
Propositional calculus "[ intro prop pyk "prop" tex "L_p" end
intro ]" as defined in \cite{mendelson} is defined thus:
"[ math theory prop end theory end math ]", "[ math in theory
prop rule a one says all meta a indeed all meta b indeed meta
a imply meta b imply meta a end rule end math ]", ...
```

to get

> Propositional calculus [$\mathrm{L_p}$] as defined in [Men87] is defined thus: [**Theory** $\mathrm{L_p}$], [$\mathrm{L_p}$ **rule** A1: $\forall\mathcal{A}$: $\forall\mathcal{B}$: $\mathcal{A} \Rightarrow \mathcal{B} \Rightarrow \mathcal{A}$], [$\mathrm{L_p}$ **rule** A2: $\forall\mathcal{A}$: $\forall\mathcal{B}$: $\forall\mathcal{C}$: $(\mathcal{A} \Rightarrow \mathcal{B} \Rightarrow \mathcal{C}) \Rightarrow (\mathcal{A} \Rightarrow \mathcal{B}) \Rightarrow \mathcal{A} \Rightarrow \mathcal{C}$], [$\mathrm{L_p}$ **rule** A3: $\forall\mathcal{A}$: $\forall\mathcal{B}$: $(\neg\mathcal{B} \Rightarrow \neg\mathcal{A}) \Rightarrow (\neg\mathcal{B} \Rightarrow \mathcal{A}) \Rightarrow \mathcal{B}$], and [$\mathrm{L_p}$ **rule** MP: $\forall\mathcal{A}$: $\forall\mathcal{B}$: $\mathcal{A} \vdash \mathcal{A} \Rightarrow \mathcal{B} \vdash \mathcal{B}$].

For a less cramped and more complete example see Section 1.6 of the body of `http://yoa.dk/logiweb/page/check/fixed/`.

The `"[ intro prop pyk "prop" tex "L_p" end intro ]"` in the source above says that the construct named "prop" should be rendered as "`L_p`" in TEX and can be referred to as "prop" on pages referencing the page. Normally, a construct should have the same name on the page and on pages referencing the page, so the latter piece of information is a bit redundant.

2.5 Proving something

Our user may now state a lemma and a proof:

> [$\mathrm{L_p}$ **lemma** I: $\forall\mathcal{A}$: $\mathcal{A} \Rightarrow \mathcal{A}$]

$\mathrm{L_p}$ **proof of** I:

L01:	Arbitrary $\gg$	$\mathcal{A}$	;
L02:	A1 $\gg$	$\mathcal{A} \Rightarrow (\mathcal{A} \Rightarrow \mathcal{A}) \Rightarrow \mathcal{A}$	;
L03:	A1 $\gg$	$\mathcal{A} \Rightarrow \mathcal{A} \Rightarrow \mathcal{A}$	;
L04:	A2 $\gg$	$(\mathcal{A} \Rightarrow (\mathcal{A} \Rightarrow \mathcal{A}) \Rightarrow \mathcal{A}) \Rightarrow$ $(\mathcal{A} \Rightarrow \mathcal{A} \Rightarrow \mathcal{A}) \Rightarrow \mathcal{A} \Rightarrow \mathcal{A}$	;
L05:	MP $\triangleright$ L02 $\triangleright$ L04 $\gg$	$(\mathcal{A} \Rightarrow \mathcal{A} \Rightarrow \mathcal{A}) \Rightarrow \mathcal{A} \Rightarrow \mathcal{A}$	;
L06:	MP $\triangleright$ L03 $\triangleright$ L05 $\gg$	$\mathcal{A} \Rightarrow \mathcal{A}$	$\square$

2.6 How the page is verified

Logiweb pages are verified by the Logiweb core software. That software would fit naturally into a Logiweb browser. But, at present, there is no such Logiweb browser since Logiweb piggybacks the World Wide Web, and the core software actually resides in the compiler, sandwiched between a frontend and a backend. When the "hello world" page is translated, the compiler does as follows:

The frontend reads the source file and loads all Logiweb pages transitively referenced by the page. During this process, all transitively referenced pages are processed and verified by the core software unless they already reside in the cache of the compiler (which may be saved to disk).

Then the compiler reads declarations of associativity and priority of all constructs used and parses the source. The output from this process is a list of bytes called a "Logiweb vector" in the format used for storing and transmitting Logiweb pages.

At this point, the compiler could store the vector and halt. Instead, the compiler invokes the core software on the vector. The core software unpacks the vector into a "body", a "bibliography", and a "dictionary". The body essentially is one, big Lisp

S-expression, the bibliography is a list of pages directly referenced by the page, and the dictionary contains the arities of all constructs introduced on the page.

Then the compiler invokes a macro engine. That engine is defined on the first, direct reference of the page, i.e. on the "base" page in the case of "hello world". The macro engine on the base page happens to implement an outside-in macro expansion strategy which supports but does not enforce recursive macro expansion. Among other, the base page contains a macro defined by [(x)≝x] which says that parentheses disappear during macro expansion.

After macro expansion, the compiler "harvests" the page, i.e. collects all definitions present on the page (including macro definitions). After harvesting, new macro definitions may affect how the page should have been expanded, so the compiler macro expands the page once more from scratch. The compiler then alternates between expansion and harvesting until a fixed point is reached (if ever) or until the user kills the compiler. The output from this process (if any) is a "codex" which contains all the definitions and an "expansion" which is the macro expanded version of the page.

Then the compiler invokes a verifier. That verifier is also defined on the first direct reference of the page. The verifier on the base page happens to be the conjunction of two verifiers, one that verifies test cases like $[2 + 3 = 5]$ and one that verifies proofs.

Since definitions is what Logiweb collects, anything interesting should be expressed as definitions. As an example, [L_p **lemma** I:$\forall\mathcal{A}$:$\mathcal{A} \Rightarrow \mathcal{A}$] macro expands into a definition that says that the "statement aspect" of [I] equals [$L_p \vdash \forall\mathcal{A}$:$\mathcal{A} \Rightarrow \mathcal{A}$]. [**Theory** L_p] is particularly complicated; it macro expands into a definition that says the the statement aspect of [L_p] equals the intuitionistic conjunction of the four rules of [L_p]. The [**Theory**] macro finds the rules of [L_p] by scanning the codex of the page.

The proof of [I] above macro expands into a definition of the "proof aspect" of [I]. The right hand side of that definition comprises a "proof engine" applied to a quoted version of the macro expanded proof. The proof verifier evaluates the right hand side so that control is passed to the proof engine which in turn evaluates all constructs for which a "tactic aspect" is defined. When all proof tactics are evaluated, the proof verifier returns a term expressed in Logiweb sequent calculus to the proof verifier which then evaluates that according to the rules of that calculus to see if the proof is correct and proves what it is supposed to prove. The proof above uses a unification tactic [$\mathcal{A} \gg \mathcal{B}$] which instantiates $\mathcal{A}$ to fit $\mathcal{B}$.

After verification, regardless of whether the page is correct or not, the compiler invokes the backend to render the body, bibliography, dictionary, codex, expansion, diagnose (in case of errors), reference and vector of the page in a number of different formats.

3 Classical reasoning about programs

Logiweb has been designed with the goal to support classical reasoning about programs. At the same time, however, Logiweb has been designed to be as neutral as possible with respect to choice of logic and notation. In other words, the requirement to support classical reasoning about programs should be seen as a widening of the scope compared to systems which focus on constructive reasoning or which focus on classical mathematics.

Constructive reasoning often leads to unnecessary complications. On the other hand, classical reasoning about programs is non-trivial because general recursion and infinite looping is cumbersome to deal with in classical theories like ZFC set theory. As an example, if $[n! \doteq \mathbf{if}(n = 0, 1, n \cdot (n-1)!)]$ then it is trivial that $[(-1)!]$ loops indefinitely, but that is non-trivial to express and prove in ZFC.

In Logiweb, the chosen solution to that problem is to base all computable definitions on a version of λ-calculus that allows classical reasoning, and to ensure that Logiweb is able to support such classical reasoning.

This is non-trivial for two reasons. Firstly, λ-calculus programs are inefficient compared to programs expressed in other languages unless special measures are taken. Secondly, it is much easier to develop first order predicate calculus in lamdba calculus than the other way round, so Logiweb must support classical logic that is not based on first order predicate calculus (we consider a theory "classical" if it allows proof by cases such as use of the law of excluded middle; the theory we shall arrive at also allows to use the axiom of choice).

In the following we first introduce a version of λ-calculus which is suited to classical reasoning and then deal with the efficiency problem.

3.1 λ-calculus for classical reasoning

Pure λ-calculus [Chu41] is inherently syntactic of nature and cumbersome to handle in classical theories like ZFC. In contrast, impure λ-calculi do support classical reasoning and have models that are classical of nature [BG97].

As an example, λ-calculus to which one adds integers is suited for classical reasoning. λ-calculus enriched with truth values T and F also supports classical reasoning.

Hence, λ-calculus enriched with two or countably many new values is suited to classical reasoning. Actually, λ-calculus enriched with any number of new values from one and up are equally suited to classical reasoning.

To make matters as simple as possible, Logiweb is based on λ-calculus enriched with just one new value plus an operation which makes the new value useful. We shall refer to the new value as T. In terms of the C programming language, a lambda construct corresponds to a pointer to a function and T corresponds to the NULL pointer. Among many things, we shall use T to represent truth, which explains the choice of name.

The λ-calculus used by Logiweb is defined by the Logiweb reduction system $\lambda\mathsf{T}$:

$$\begin{array}{lcl} (\lambda x.y)z & \rightarrow & \langle y|x{:=}z\rangle \\ \mathsf{T}z & \rightarrow & \mathsf{T} \\ \mathbf{if}(\mathsf{T}, y, z) & \rightarrow & y \\ \mathbf{if}(\lambda u.v, y, z) & \rightarrow & z \end{array}$$

3.2 Equality of lambda terms

In pure λ-calculus, two terms are considered "equal" or "beta-equivalent" if they reduce to the same term; this is what makes λ-calculus syntactic of nature and cumbersome to deal with classically. In $\lambda\mathsf{T}$, equality is defined semantically as follows:

Define $\lambda\mathsf{T}\|$ as the system above extended with a new, binary operator $x \parallel y$ and the following reduction rules:

$$\begin{array}{lcl} \mathsf{T} \parallel x & \to & \mathsf{T} \\ x \parallel \mathsf{T} & \to & \mathsf{T} \\ \lambda u.v \parallel \lambda x.y & \to & \lambda z.\mathsf{T} \end{array}$$

We say that a term t of $\lambda\mathsf{T}\|$ is a "true" term if it reduces to T, a "function" term if it reduces to a term of form $\lambda x.y$, and a "bottom" term if it is neither a true nor a function term. We say that two terms s and t are "root equivalent", written $s \sim t$, if they are both true, both function, or both bottom terms. We say that s and t are "equal", written $s = t$, if $fs \sim ft$ for all terms f of $\lambda\mathsf{T}\|$. Terms of $\lambda\mathsf{T}$ are considered equal if they are equal in $\lambda\mathsf{T}\|$.

Classical equality $s = t$ is undecidable (since otherwise $s = \bot$ could decide the halting problem where $\bot \doteq (\lambda x.xx)(\lambda x.xx)$). Furthermore, $s = t$ differs from $\beta\eta$-equivalence $s =_\eta t$. Actually, neither of the two relations imply the other. As an example, $(\lambda x.\lambda y.xy)\mathsf{T} \to \lambda y.\mathsf{T}y$ whereas $(\lambda x.x)\mathsf{T} \to \mathsf{T}$ so $\lambda x.\lambda y.xy \neq \lambda x.x$ even though $\lambda x.\lambda y.xy =_\eta \lambda x.x$. As another example, $\mathsf{Y}\lambda f.\lambda x.\lambda x.f = \mathsf{Y}\lambda f.\lambda x.\lambda x.\lambda x.f$ (they both equal $\lambda x.\lambda x.\lambda x.\cdots$) but the two terms are not $\beta\eta$-equivalent. β-equivalence does imply classical equality.

3.3 Classical reasoning about $\lambda\mathsf{T}$

In $\lambda\mathsf{T}$, any term f satisfies $f = \mathsf{T}$ or $f = \lambda x.fx$ or $f = \bot$, there is no fourth possibility. This 'quartum non datur' (QND) rule makes $\lambda\mathsf{T}$ classical because it allows proof by cases. A two-valued logic like ordinary propositional calculus satisfies a 'tertium non datur' rule whereas a three valued logic like $\lambda\mathsf{T}$ satisfies a 'quartum non datur' rule like the one above. The proof-theoretic strength is the same.

As an example, if we define $\mathsf{F} \doteq \lambda x.\mathsf{T}$ and $x \wedge y \doteq \mathbf{if}(x, \mathbf{if}(y, \mathsf{T}, \mathsf{F}), \mathbf{if}(y, \mathsf{F}, \mathsf{F}))$ then QND allows to prove $x \wedge y = y \wedge x$. If $x \vee y \doteq \mathbf{if}(x, \mathbf{if}(y, \mathsf{T}, \mathsf{T}), \mathbf{if}(y, \mathsf{T}, \mathsf{F}))$ and $\neg x \doteq \mathbf{if}(x, \mathsf{F}, \mathsf{T})$ then $x \vee \neg x = \mathsf{T}$ fails (counterexample: $x = \bot$) but QND allows to prove $x \vee \neg x = !x$ where $!x \doteq \mathbf{if}(x, \mathsf{T}, \mathsf{T})$. In general, QND allows to translate lemmas and proofs of classical propositional calculus to $\lambda\mathsf{T}$.

The QND-inference belongs to Map Theory. In Logiweb, Map Theory has four connections to $\lambda\mathsf{T}$. Firstly, Map Theory allows to reason about $\lambda\mathsf{T}$. Secondly, the reduction rules of $\lambda\mathsf{T}$ are axioms of Map Theory. Thirdly, any $\lambda\mathsf{T}$ program like $x! \doteq \mathbf{if}(x = 0, 1, n \cdot (n-1)!)$ (where integers and $=$ and $\cdot$ on integers is defined suitably) automatically becomes a definition that can be used in Map Theory proofs. Fourthly, the proof checker for Map Theory is implemented in $\lambda\mathsf{T}$.

3.4 Stress test of Map Theory

The ultimate test for a theory is to prove the consistency of ZFC set theory within it. The result itself is not important since ZFC set theory is generally accepted to be consistent, but proving the consistency of ZFC in a theory proves that that theory is as powerful as ZFC theory which in turn is known to be sufficiently powerful to express all of classical and most of modern mathematics.

For Map Theory, [Gru02] contains a formal proof of the consistency of ZFC within Map Theory. Among other, [Gru02] was written to develop the language of Logiweb, so

[Gru02] was expressed in the language of Logiweb before Logiweb was implemented. The correctness of [Gru02] has been established in Isabelle as reported in [Ska02]. Adaption of [Gru02] to the final version of the Logiweb language and proof-checking in that framework is the next, major task in the Logiweb project and will allow comparison with the Isabelle implementation.

Note: If SI denotes the assumption that there exists a strongly inaccessible ordinal then ZFC+SI can prove the consistency of Map Theory [BG97, Gru92] which in turn can prove the consistency of ZFC [Gru92, Gru02]. This supports a claim that Map Theory has strength between ZFC and ZFC+SI. But "strength" is defined on basis of Gödels 1931 paper [Göd31] which only considers theories that build on first order predicate calculus so, for technical reasons, the "strength" of Map Theory is not defined. The claim in Section 1 that Map theory has the "same" power as ZFC is imprecise but close to the truth.

3.5 Efficiency of $\lambda\mathsf{T}$

The core software of Logiweb supports $\lambda\mathsf{T}$ and no other programming language. Since classical reasoning about $\lambda\mathsf{T}$ is possible and rather straightforward ([Gru02], Chapter 6), this ensures the possibility to reason classically about any program expressed in Logiweb.

This leaves two problems: How to handle programs expressed in other languages, and how to ensure efficiency?

The first problem is trivial in principle. To handle e.g. C programs, define a compiler from C to $\lambda\mathsf{T}$ in $\lambda\mathsf{T}$. Such a compiler is typically referred to as a 'semantics' of C.

The second problem is non-trivial, and typical implementations of λ-calculus are inefficient to a degree where a compiler from C to $\lambda\mathsf{T}$ would be of little practical use.

Logiweb has a rather simple solution to the efficiency problem which is described in the following.

3.6 Definitions

Logiweb allows Logiweb pages to contain definitions. As an example, consider the following definitions:

$$\begin{array}{lcl} x::y & \doteq & \lambda z.\mathbf{if}(z,x,y) \\ x^h & \doteq & x\mathsf{T} \\ x^t & \doteq & x\mathsf{F} \\ \mathsf{F} & \doteq & \lambda x.\mathsf{T} \end{array}$$

It is straightforward to prove $(x::y)^h = x$ and $(x::y)^t = y$ so $x::y$ is a pairing construct and x^h and x^t are the associated 'head' and 'tail' operations.[1]

If the definitions above are stated on a Logiweb page P, then they will be available in P as well as all pages referencing P. The definitions effectively extend the Logiweb reduction system with new reduction rules like $x::y \rightarrow \lambda z.\mathbf{if}(z,x,y)$.

[1] In Map Theory, which has inference rules of transitivity and substitution of equals and which has all $\lambda\mathsf{T}$ reduction rules and all valid $\lambda\mathsf{T}$ definitions as axioms, a proof of $(x::y)^h = x$ essentially reads $(x::y)^h = (x::y)\mathsf{T} = (\lambda z.\mathbf{if}(z,x,y))\mathsf{T} = \mathbf{if}(\mathsf{T},x,y) = x$.

Computation of e.g. $(x::y)^h = x$ will not be particularly efficient, however. It is possible to implement the pairing operation much more efficiently than using $\lambda z.\mathbf{if}(z, x, y)$.

To allow efficient implementation without affecting the ability of classical reasoning, Logiweb has two definition constructs, which we shall refer to as $\doteq$ and $\doteqdot$. Formally,

$$x::y \doteq \lambda z.\mathbf{if}(z, x, y)$$

and

$$x::y \doteqdot \lambda z.\mathbf{if}(z, x, y)$$

mean exactly the same. Backstage, however, Logiweb has a finite list of 'optimized functions' which Logiweb can compute efficiently. For each optimized function, Logiweb stores both the efficient version of the function and the 'semantics' of the function. The 'semantics' of an optimized function is a definition of the function expressed in $\lambda\mathsf{T}$. When Logiweb sees a definition like

$$x::y \doteqdot \lambda z.\mathbf{if}(z, x, y)$$

it searches its list of optimized functions for one whose 'semantics' is identical to

$$\lambda z.\mathbf{if}(z, x, y)$$

(except for naming of bound variables). If Logiweb finds a match, it translates $x::y$ to the optimized function found. Otherwise, Logiweb treats $\doteqdot$ like $\doteq$. A "match" must be exact (modulo naming). As an example, $\lambda z.\mathbf{if}(z, (\lambda x.x)x, y)$ does not match $\lambda z.\mathbf{if}(z, x, y)$.

The $\doteq$ and $\doteqdot$ constructs are identical from the point of view of reasoning as long as optimized functions behave exactly as specified by their semantics. It is the responsibility of the implementer of the core software to ensure this correspondence.

Distinct implementations of Logiweb may have different lists of optimized functions; that may affect the speed of a computation but not the result of a computation.

Actually, the current implementation of Logiweb has no optimized function for the untagged pair construct $x::y$ above. Instead, that implementation has an optimized function for a particular tagged pair construct, and also has optimized functions for handling cardinals (i.e. non-negative integers). Finally, the current implementation of Logiweb does some type inference and strictness analysis to make programs run fast (c.f. Section 3.6 of the base page). All that is invisible from the point of view of reasoning about programs.

3.7 Feasibility

The measures above and those mentioned in Section 4 have proven sufficient to make it feasible to implement macro expansion and proof checking on top of $\lambda\mathsf{T}$. As mentioned in the abstract, the 'base' page on `http://yoa.dk/`, which is 180 pages long when printed out, was macro expanded and checked in 40 seconds.

Each time an efficiency enhancement was implemented, the efficiency gain was measured in a rather crude way (with a manual stop watch on an otherwise unloaded

machine). The product of efficiency gains indicate a total speed-up around 10^9 with an uncertainty of several orders of magnitude. If the 10^9 figure is correct, the base page would take around 1200 years to macro expand and check without optimizations.

The 40 second measure is just a feasibility demonstration, not one suited for comparison with other systems. The measure shows that Logiweb can survive a 180 page document with around 1500 definitions, 180 test cases, and 10 proof lines. The 40 seconds are mainly spent on macro expanding the rather complex base page seven times (for macro expansion iteration see Section 2.6). Applying Logiweb to longer proofs is currently done by about ten students on a graduate course.

4 Data structures

In this section we describe the data structures Logiweb uses when verifying pages. The choice of such data structures has proven to impact the efficiency of verification considerably.

4.1 Terms

The current implementation of Logiweb has a pairing function and support for cardinals (non-negative integers) among its optimized functions. We shall refer to the pairing function as $x::y$ even though it differs from the particular pairing function defined in Section 3.

The term is one of the most fundamental data structures of a system for formal logic. Logiweb terms are data structures implemented using cardinals, $x::y$, and T.

Logiweb terms are used for representing ordinary terms like $2+3$. But they are also used for representing formulas like $\forall x\colon x+1=1+x$. Furthermore, Logiweb terms are used for representing lemmas and proofs. Actually, an entire Logiweb page is one big term as seen from the point of view of Logiweb.

Logiweb terms are trees whose nodes are labeled by 'Logiweb symbols'. A Logiweb term with root r and subterms $t_1,\ldots,t_n$ is represented by

$$r::t_1::\cdots::t_n::\mathsf{T}$$

This representation of terms is effectively the same as a Lisp S-expression [McC60]. The differences are (1) Logiweb symbols differ from Lisp symbols/Lisp atoms, (2) Logiweb terms are terminated by T where Lisp S-expressions are terminated by NIL, and (3) each Logiweb symbol has an arity which must match the n above.

As mentioned previously, each Logiweb page has a reference r. That reference is a sequence of bytes when stored on disk or transmitted via a network, but when handled inside Logiweb software, it is a cardinal.

Each Logiweb page declares a finite number of Logiweb symbols. Each Logiweb page has a unique reference r and each symbol declared by a page has an identifier i which is unique within the page, so a Logiweb symbol is uniquely determined by r and i together.

A Logiweb symbol with reference r and identifier i is represented by a structure of form $r::i::d$. The last item d in a Logiweb symbol comprises debugging information

which is irrelevant to formal reasoning. The debugging information notes where the term was located before macro expansion and thus allows to produce meaningful error messages.

The term that makes up a Logiweb page can only contain symbols from the page itself and pages directly referenced by the page. After macro expansion, the term can contain symbols from the page itself and pages transitively referenced by the page.

Large parts of a Logiweb page typically consists of ordinary text. Ordinary text is encoded as terms inside the Logiweb software. When stored on disk or transmitted over a network, care has been taken to encode text efficiently. In particular, Unicode characters below 128 (i.e. ASCII characters) take up one byte each. Ordinary text of Logiweb pages is expressed as TEX source text.

4.2 Arrays

Concerning efficiency, the main drawback of pure functional programming is the lack of constant time array access. Logiweb is based on $\lambda\mathsf{T}$ which certainly is a pure functional programming language, and constant time array access is not tenable. Fortunately, it is not important whether or not array access is constant in time. It is more important that array access is fast.

Apart from terms, the main data structures of Logiweb are based on 'Logiweb arrays'. A Logiweb array a represents a function f from cardinals to arbitrary data which has the property that $f(n) \neq \mathsf{T}$ holds for at most finitely many cardinals n.

We shall refer to the value associated to the cardinal n by the array a as $a[n]$ and to the set $\{n|a[n] \neq \mathsf{T}\}$ as the 'domain' of a.

A Logiweb array a is represented as a binary tree whose leafs have form $n::x$ where $x \neq \mathsf{T}$. A leaf of form $n::x$ represents the information that $a[n] = x$.

A leaf of form $n::x$ is placed at a location in a which depends on the index n and on what other indices are stored in the array. As an example, consider a leaf of form $6::\mathsf{F}$ which indicates that $a[6] = \mathsf{F}$. The binary expansion of 6, written with the least significant bit first, reads $01100000\cdots$. The address at which the leaf $6::\mathsf{F}$ is placed in a is the shortest prefix of $01100000\cdots$ which distinguishes 6 from all other indices of the array a.

As a result, the access time of an array a with a contiguous domain depends on the logarithm of the size of the domain. The access time of a sparse array with randomly distributed indices also depends on the logarithm of the size of the domain.

The arrays used in Logiweb are typically accessed either by small cardinals (e.g. identifiers of Logiweb symbols) or by randomly distributed cardinals (e.g. references of Logiweb symbols).

4.3 Logiweb codices

A definition like $\mathsf{F} \doteq \lambda x.\mathsf{T}$ defines the value of F, and a system for mathematical reasoning certainly must keep track of such definitions. Logiweb collects all definitions present on a Logiweb page in a data structure which we shall refer to as a Logiweb codex, c.f. Section 2.6.

A value definition like $\mathsf{F} \doteq \lambda x.\mathsf{T}$ is what one normally thinks of as a definition. But a computational system must handle many other kinds of definitions: definitions of how

constructs macro expand, how they should be rendered, how a user should input them via a keyboard, and many other things.

Logiweb handles different kinds of definitions by the introduction of Logiweb 'aspects'. Each definition in Logiweb consists of a left hand side, a right hand side, and an aspect. As an example, definition of three aspects of F could read

$$\begin{aligned}\mathsf{F} &\overset{\text{val}}{=} \lambda x.\mathsf{T}\\ \mathsf{F} &\overset{\text{tex}}{=} \text{“\mathsf\{F\}”}\\ \mathsf{F} &\overset{\text{pyk}}{=} \text{“false”}\end{aligned}$$

The first definition defines the value aspect of F. Or, stated in a more straightforward way, it defines the value of F. The second definition defines how F should be rendered and the third definition states what a user should type on a keyboard or say in a microphone to enter an F to an authoring tool. On traditional, site based proof checkers one typically stores "pyk"- and "tex"-like information separately, but when sending pages around on the internet, each page must be a capsule containing all information needed to e.g. render the page. The latter two definitions above form a convenient way to include information that is normally hidden away. The "intro" construct in Section 2.4 macro expands into pyk and tex definitions (see the base page for a precise definition).

The macro facility allows to keep the de Bruijn factor[2] low. The macro facility allows authors to write pages in a style that is appealing to the human reader but still macro-reduces into a more machine understandable form. The author of a page can define a construct to be a macro by defining its macro aspect.

In Logiweb, aspects are represented by symbols. Definitions contain a left hand side (which may contain parameters), an aspect (which may also contain parameters), and a right hand side. When Logiweb "codifies" a Logiweb page, it macro expands it and collects definitions from it, leading to a "codex" and a macro expanded version of the page. The macro expanded version is a term whereas the codex is an array c with the property that

$$c[r_s][i_s][r_a][i_a]$$

is the definition of the aspect with reference r_a and identifier i_a for the symbol with reference r_s and identifier i_s.

The codex allows fast access to any aspect of any symbol during verification.

4.4 Logiweb racks

We shall refer to an array that contains heterogeneous data as a 'rack' and to each index of a rack as a 'hook'.

Logiweb assigns a rack r to each Logiweb page. The hooks of the rack are various cardinals that represent various concepts. As an example, Logiweb hangs the codex c of a page on the 'codex' hook, meaning that a particular cardinal c' represents the concept of a codex and meaning that $r[c'] = c$. The hooks of a Logiweb rack include the following:

[2] the ratio by which formalization increases the length of a text [dB80]

vector The list of bytes that makes up the Logiweb page when it is stored on disk or transmitted over a network.

bibliography The list of pages directly referenced by the present page. Reference number zero (the first reference) is the reference of the page itself.

cache Explained later.

dictionary Symbol declarations of the page, represented as an array d. A symbol with identifier i 'exists' if $d[i] \neq \mathsf{T}$ in which case $d[i]$ is the arity of the symbol.

body The term that makes up the page.

codex The codex of the page as explained previously.

expansion The macro expanded version of the body.

diagnose Logiweb hangs T on this hook if the page passes verification. Otherwise, the diagnose will be a term which, when typeset, is supposed be be a meaningful error message.

code A compiled version of the codex (for an example of use see the base page, Section 4.4.1)

4.5 Logiweb caches

As explained previously, there is a one-to-one correspondence between Logiweb references and immutable Logiweb pages. The correctness of a Logiweb page only depends on the contents of the page, which is immutable, and the contents of transitively referenced pages, which are also immutable. For that reason, each Logiweb page only needs to be verified once. The current implementation of the compiler verifies each page the first time it is referenced within a session.

Independently of any caches maintained by Logiweb software for efficiency reasons, Logiweb also defines a "Logiweb cache" for each Logiweb page. The cache of a page collects information about a page and all its transitively referenced pages.

The cache of a Logiweb page is an array c for which $c[r]$ is the rack of the page with Logiweb reference r. The domain of the cache c comprises the references of the page itself and pages transitively referenced by the page. As a dirty trick, $c[0]$ contains the reference of the page itself so that e.g. $c[c[0]]$ is the rack of the page.

Racks and caches are defined mutually recursively. A cache is an array that maps references to racks. A rack maps the previously mentioned 'cache' hook to an array which maps references of transitively referenced pages to their caches. The resulting structure is a non-cyclic one with considerable sharing which gives efficient access to all data needed during verification and during all other activities undertaken by Logiweb.

5 Verification

5.1 Page symbols

Each Logiweb page implicitly declares a symbol whose identifier is zero, and the arity of that symbol is forced to be zero. We shall refer to that symbol as the 'page symbol' of the page. The reference of a page symbol equals the reference of the page so there is a one-to-one correspondence between pages and page symbols, c.f. Section 4.1.

Each Logiweb definition assigns an aspect to a symbol. Aspects assigned to a page symbol, however, should be interpreted as aspects of the page. As an example, the name of a page symbol effectively becomes the name of the page.

5.2 Verification

From the point of view of the Logiweb core software, verification of a Logiweb page is trivial. The core software just codifies the page, looks up the 'claim' aspect of the page symbol (which, if defined, is a term), applies that term to the cache of the page, and considers the page correct if the result is T. Otherwise, Logiweb hangs the result of the computation on the 'diagnose' hook of the page. The diagnose is supposed to be a term which, when typeset, is supposed to explain what went wrong. Supplying meaningful diagnoses is the responsibility of programmers of claims, proof tactics, etc.

If a page makes no claim (i.e. if no claim aspect is defined for its page symbol) then Logiweb uses the claim of reference number one of the page, and if that reference makes no claim either, then the page is considered trivially correct. In the "hello world" example in Section 2.6, the "hello world" page makes no claim but reference number one (the base page) does make a claim.

The claim made by the base page is a substantial one. It scans the codex of the page for all proof definitions, invokes proof compilers which in turn invoke proof tactics, verifies the proofs, checks for cyclic references between proofs, and checks that the proofs prove the statement aspects they are supposed to prove.

It is an important feature of Logiweb that a complex beast like a proof checker is not included in the core software. Firstly, it reduces the complexity of the core software. Secondly, at gives the users of the system the flexibility to use the proof checker that comes with logiweb (simply by referencing the 'base' page as reference number one) or to define another one.

To establish confidence in the formal correctness of a Logiweb page, a human reader can check that it has been verified by a proof checker that the reader trusts. That can be done by inspecting the claim aspect of the relevant page symbol.

Proof checkers are faced with the same problem as the human reader. The proof checker that comes with Logiweb is 'arrogant' in the sense that it only trusts lemmas that it has checked itself. When a proof being checked references a lemma on another page, the proof checker looks up the claim of the other page, which is supposed to be a conjunction, and then checks that the proof checker itself is a member of that conjunction. The proof checker also checks that the diagnose of referenced pages equals T. Hence, the proof checker that comes with Logiweb only trusts itself but is willing to coexist with other checkers. On the base page, the proof checker coexists with a test case verifier, c.f. Section 2.6.

One can easily adapt any TEX source to the format of Logiweb and get it accepted as a trivially correct page, simply because TEX sources make no claims that Logiweb can understand. That is useful for communication to readers but of course such trivially correct pages are of no formal use.

Macro expansion is just as simple as verification from the point of view of the Logiweb core software. Logiweb pages are macro expanded by applying the macro aspect of the page symbol of a page to the body of the page and hanging the result on the 'expansion'

hook of the page, c.f. Section 2.6..

6 Status

Logiweb allows users to publish, verify, retrieve, and read pages that contain formal mathematics.

The Logiweb core software comprises about 900 kilobyte of source code (including comments). It implements the features described in the present paper. It also implements many other features like features for rendering. The core software is kept simple by moving essential features like the definition of the proof checker from the core software to Logiweb pages.

In particular, the measures taken to allow λ-calculus programs to be efficient have been implemented with success. Also, the data structures of codices, racks, and caches have proven to support proof checking well.

At the time of writing, Logiweb allows referencing within a single site covered by a single Logiweb server. The Logiweb protocol allows cooperation among Logiweb servers. When that is implemented, the web-part of the system will allow a single formal development to consist of papers that reside different places in the world. Until then, users are forced to copy referenced papers to their own site. Section 2.3 describes copying as a convenient possibility, but until further copying is a necessity.

At the time of writing, 600 kilobyte of Logiweb source text has been verified by Logiweb. Those 600 kilobyte define the computing machinery, the macro expansion facility, and the proof checker, and verify the feasibility of the system (c.f. Section 3.7). 800 kilobyte of formal proofs [Gru02] await verification.

References

[BG97] C. Berline and K. Grue. A κ-denotational semantics for Map Theory in ZFC+SI. *Theoretical Computer Science*, 179(1–2):137–202, June 1997.

[CAB+86] Robert L. Constable, 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.

[Chu41] A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.

[dB80] Nicolaas Govert de Bruijn. A survey of the project AUTOMATH. In J.P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, 1980.

[DBP96] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. RIPEMD-160: A strengthened version of RIPEMD. In *Fast Software Encryption*, pages 71–82, 1996. `http://citeseer.nj.nec.com/dobbertin96ripemd.html`.

[Göd31] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 12(XXXVIII):173–198, 1931.

[Gru92] K. Grue. Map theory. *Theoretical Computer Science*, 102(1):1–133, July 1992.

[Gru01] K. Grue. *Mathematics and Computation*, volume 1–3. DIKU (lecture notes), 7. edition, 2001. `http://www.diku.dk/~grue/papers/mac0102/`.

[Gru02] K. Grue. Map theory with classical maps. Technical Report 02/21, DIKU, 2002. `http://www.diku.dk/publikationer/tekniske.rapporter/2002/`.

[Gru04] Klaus Grue. Logiweb. In Fairouz Kamareddine, editor, *Mathematical Knowledge Management Symposium 2003*, volume 93 of *Electronic Notes in Theoretical Computer Science*, pages 70–101. Elsevier, Feb 2004.

[Knu83] D. Knuth. *The TeXbook*. Addison Wesley, 1983.

[Koh03] Michael Kohlhase. OMDoc: An open markup format for mathematical documents (version 1.1), 2003. `http://www.mathweb.org/omdoc.ps`.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, pages 184–195, 1960.

[Men87] E. Mendelson. *Introduction to Mathematical Logic*. Wadsworth and Brooks, 3. edition, 1987.

[MS01] Robert Miner and Jeff Schaeffer. A gentle introduction to MathML, 2001. `http://www.dessci.com/en/support/tutorials/mathml/default.htm`.

[Muz93] Michał Muzalewski. *An Outline of PC Mizar*. Foundation of Logic, Mathematics and Informatics, Mizar User Group, Brussels, 1993.

[Pau98a] Lawrence C. Paulson. Introduction to Isabelle. Technical report, University of Cambridge, Computer Laboratory, 1998.

[Pau98b] Lawrence C. Paulson. The Isabelle reference manual. Technical report, University of Cambridge, Computer Laboratory, 1998.

[Ska02] Sebastian C. Skalberg. *An Interactive Proof System for Map Theory*. PhD thesis, University of Copenhagen, October 2002. `http://www.mangust.dk/skalberg/phd/`.

[TB85] Andrzej Trybulec and Howard Blair. Computer assisted reasoning with MIZAR. In Aravind Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28, Los Angeles, CA, August 1985. Morgan Kaufmann. `http://www.mizar.org/`.

[Val03] Thierry Vallée. *"Map Theory" et Antifondation*, volume 79 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.