

Simptcheck Manual

Chad E. Brown

April 13, 2008

Contents

Chapter 1. Introduction	5
1. Command Line Options	5
2. What is a Proof Checker?	6
3. What Theorems Can Simptcheck Prove Automatically?	6
Chapter 2. Syntax	7
Chapter 3. The Pure Type System	11
Chapter 4. A Signature for First Order Set Theory	13
1. First-Order Logic with Equality	13
2. A Description Operator	14
3. Set Theory	14
Chapter 5. Organization of the Code	17
1. Lexer and Parser	17
2. Syntax	17
3. State	17
4. Checker	18
Bibliography	19

Introduction

Simptcheck¹ is a simple proof checker written in OCaml. Simptcheck is designed to separate the formulation of a problem from the formulation of its solution. The type theory of Simptcheck is simply typed λ -calculus with a special base type $*$ of “internal” propositions and a sort *Prop* of “external” propositions. This is a weak type theory, but expressive enough to allow one to formulate very strong theories (e.g., ZFC set theory) as signatures.

Here is how simptcheck is intended to be used:

- (1) User **A** writes files

$$file_1, \dots, file_n$$

specifying an underlying theory and a problem containing some missing definitions d_1, \dots, d_m and open claims c_1, \dots, c_k . Simptcheck can check that the problem is well-formed as follows:

```
simptcheck file_1 ... file_n
```

- (2) User **B** can either provide files giving a solution or proving the problem is inconsistent (has no solution).

- (a) User **B** provides a solution by giving files

$$file_{n+1}, \dots, file_{n+p}$$

which gives terms defining d_1, \dots, d_m and proof terms for the claims c_1, \dots, c_k . The new files cannot contain any new constants or axioms. User **A** can use simptcheck to check that the new files form a solution as follows:

```
simptcheck file_1 ... file_n -s file_{n+1} ... file_{n+p}
```

- (b) User **B** shows the problem has no solution by giving files

$$file_{n+1}, \dots, file_{n+p}$$

which proves false by treating d_1, \dots, d_m as constants and c_1, \dots, c_k as axioms. The new files cannot contain any new constants or axioms. User **A** can use simptcheck to check that the new files form a proof of inconsistency as follows:

```
simptcheck file_1 ... file_n -n file_{n+1} ... file_{n+p}
```

In each case, simptcheck exits with 0 upon success and 1 upon failure.

1. Command Line Options

We summarize the command line options:

- **-V**: Be verbose.

¹Copyright Chad E. Brown 2008

- `-r`: Upon success, give a final report giving the number of missing definitions, open claims and proven claims.
- `-i`: Read from `stdin` instead of from files.
- `-h` or `-help`: Print a help message and exit.
- `-s`: Check that the files after `-s` form a solution.
- `-n`: Check that the files after `-n` form a proof of inconsistency.

2. What is a Proof Checker?

A proof checker is a program that given a proposition P and an alleged proof of P will check if the alleged proof conforms to the rules of the underlying logic and proves P . Using the Curry-Howard correspondence (see, e.g., [6]), one can reduce proof checking to type checking in an appropriate type theory [1, 2, 10]. Some of the rules of the underlying logic are built into the type theory. The remaining rules are encoded in the “signature” – an ordered set of constants (associated with types) and abbreviations (associated with types and terms). (Warning: In some type theories the distinction between *types* and *terms* becomes blurred.)

The original proof checker of this kind was Automath [12]. One of the great success stories of formalized mathematics (in my opinion, at least) was the formalization by Jutting [8] of Landau’s *Grundlagen der Analysis* [9] in Automath during the 1970’s. This included a construction of the real numbers (as Dedekind cuts) starting from the natural numbers (with the Peano axioms) and a bit of higher-order logic. I should note that Automath (as a type theory) is unusual when compared to modern type theories, as λ is the only binder. Two modern type checking/proof checking systems are Coq [3] and Twelf [11]. Both of these systems include far more features than mere type checking/proof checking.

Simptcheck is like Automath with a particularly weak, but modern, type theory. The weakness of the type theory allows one to be semantically flexible. As discussed in my book [4], simple type theory has many unusual models if one is flexible about which logical constants must be realized. If we move logical constants from the type theory (where they would correspond to a Π binder or an infix \Rightarrow) to the signature, then we allow for signatures in which the corresponding logical constants may be missing. For some theories, this is important. For example, one can encode first-order set theory into higher-order logic as in [7], but the higher-order quantifiers strengthen the theory. If we remove all logical constants, then we can encode first-order set theory in simple type theory by declaring only the *first-order* logical constants and then giving the set theory axioms.

3. What Theorems Can Simptcheck Prove Automatically?

None. Simptcheck is a proof checker, not an automated theorem prover. The intention is to have Simptcheck check a formal proof after it has been constructed (whether automatically, interactively, or in some combination of these modes).

CHAPTER 2

Syntax

The following are reserved words:

- `var`
- `hyp`
- `const`
- `def`
- `abstract`
- `abstract_all`
- `known`
- `claim`
- `proof`
- `accept`
- `accept_all_claims`
- `inconsistent`
- `*`

An identifier *id* is a nonempty sequence of characters from the set

$$\{_, +, *, ', -, 0 - 9, a - z, A - Z\}$$

unless the sequence forms a reserved word. We write *ids* to mean a nonempty sequence of identifiers separated by whitespace.

There are four categories of syntactic objects: *types*, *terms*, *props* and *proofs*. We give the grammar for each of these categories in Figure 1.

We specify problems and solutions by giving a sequence of document items in files. The list of possible document items is given in Figure 2. Below we describe the meaning of the document items in more detail.

- `var ids: type;`
After this declaration, whenever the listed names are used as variables bound by the term binder `\` and the type of the variable cannot be determined, the type given here is used.
- `hyp id: prop;`
After this declaration, whenever the name is used in a proof binder `/` and the prop cannot be determined, the prop given here is used.
- `const id: type;`
Add a constant to the signature. This is not allowed in solutions or inconsistency proofs.
- `def id: type;`
Add the name with the type to the signature. For proof checking purposes the name is treated as a constant. Unlike constants, the name can later be given a definition by giving a term of the given type.

<i>type</i>	::=	<i>id</i>	(name of a base type)
		*	(base type of atomic propositions)
		<i>type</i> > <i>type</i>	(function type)
<i>term</i>	::=	<i>apterm</i>	(application term)
		\ <i>ids</i> . <i>term</i>	(lambda abstraction)
		\ <i>ids</i> : <i>type</i> . <i>term</i>	(lambda abstraction with domain)
		(<i>term</i>)	
<i>apterm</i>	::=	<i>id</i>	(basic term)
		<i>apterm</i> <i>id</i>	(application to a basic term)
		<i>apterm</i> (<i>term</i>)	(application to a term)
<i>prop</i>	::=	<i>term</i>	(atomic proposition, must have type *)
		<i>apterm</i> > <i>prop</i>	(implication)
		(<i>prop</i>)> <i>prop</i>	(implication)
		! <i>ids</i> . <i>prop</i>	(universal quantifier)
		! <i>ids</i> : <i>type</i> . <i>prop</i>	(universal quantifier with domain)
<i>proof</i>	::=	<i>aproof</i>	(application proof term)
		\ <i>ids</i> . <i>proof</i>	(lambda abstraction)
		\ <i>ids</i> : <i>type</i> . <i>proof</i>	(lambda abstraction with domain)
		/ <i>ids</i> . <i>proof</i>	(proof lambda abstraction)
		/ <i>ids</i> : <i>prop</i> . <i>proof</i>	(proof lambda abstraction with domain)
		(<i>proof</i>)	
<i>aproof</i>	::=	<i>id</i>	(axiom or claim)
		<i>aproof</i> <i>id</i>	(application to a basic term, axiom or claim)
		<i>aproof</i> (<i>term</i>)	(application to a term)
		<i>aproof</i> (<i>proof</i>)	(application to a proof)

FIGURE 1. Syntax for types, terms, props and proofs

- **def** *id*: *type*=*term*;
Add the name to the signature as a definition. Definitions are expanded during proof checking.
- **def** *id*=*term*;
If the identifier corresponds to a missing definition, then this checks if the term has the expected type and (if so) fills in the missing definition with the term. If the identifier does not correspond to a missing definition (and the term is well-typed) then it is added to the signature as a new definition.
- **abstract** *ids*;
After abstracting definitions, they are no longer expanded during proof

<code>var <i>ids</i>:<i>type</i>;</code>	(declare a default type for the names)
<code>hyp <i>id</i>:<i>prop</i>;</code>	(declare a default prop for the name)
<code>const <i>id</i>:<i>type</i>;</code>	(declare a constant of the given type)
<code>def <i>id</i>:<i>type</i>;</code>	(declare a missing definition of the given type)
<code>def <i>id</i>:<i>type</i>=<i>term</i>;</code>	(declare a definition of the given type)
<code>def <i>id</i>=<i>term</i>;</code>	(fill a missing definition or declare a new definition)
<code>abstract <i>ids</i>;</code>	(stop expanding the given definitions during checking)
<code>abstract_all;</code>	(stop expanding all previous definitions during checking)
<code>known <i>id</i>:<i>prop</i>;</code>	(declare a prop as true)
<code>claim <i>id</i>:<i>prop</i>;</code>	(declare an open claim)
<code>accept <i>ids</i>;</code>	(accept the listed open claims and missing definitions)
<code>accept_all_claims;</code>	(accept all previous open claims and missing definitions)
<code>proof <i>id</i>=<i>proof</i>;</code>	(give a proof of an open claim)
<code>inconsistent <i>proof</i>;</code>	(give a proof of false)

FIGURE 2. Syntax for document items

checking. It is important to abstract definitions or proof checking will become too time-consuming. One can always declare a definition, prove claims corresponding to folding and unfolding the definition, abstract the definition, and then use the folding and unfolding results when necessary. In some cases (e.g., an implementation of ordered pairs in set theory) we may want to forget the definition altogether after we have proven the fundamental properties of the definition.

- `abstract_all;`
Abstract all previous definitions.
- `known id:prop;`
Declare a prop to be known, i.e., an axiom. This is not allowed in solutions.
- `claim id:prop;`
Declare an open claim. This will either be proven (using `proof`), accepted (using `accept` or `accept_all_claims`), or left open for someone to prove later.
- `accept ids;`
Accept the listed open claims (essentially as axioms) and missing definitions (as constants). This is not allowed in solutions.
- `accept_all_claims;`
Accept all open claims and missing definitions. This is not allowed in solutions. The primary intended use of this is when the problem contains a number of claims which we have proven, but do not want to recheck the proofs when we want to check a proposed solution.

- `proof id=proof;`
Give a proof of an open claim. The proof term is checked to ensure it has the prop corresponding to the open claim.
- `inconsistent proof;`
Give a proof of “false,” i.e., a proof term of type $!p : *.p$ (or, in the usual type theory notation, $\Pi p : *.p$). If we can prove this, then we have an inconsistent signature. Technically, these means every proposition is provable relative to the signature.

The Pure Type System

The checker checks the pure type system (PTS) (see [1, 2]) with four sorts

$$Kind, Type, Prop, *$$

three axioms

$$Type : Kind$$

$$* : Type$$

$$Prop : Kind$$

and seven rules

$$(Type, Type, Type)$$

$$(Type, *, Prop)$$

$$(Type, Prop, Prop)$$

$$(*, *, Prop)$$

$$(*, Prop, Prop)$$

$$(Prop, *, Prop)$$

$$(Prop, Prop, Prop).$$

The sort $*$ is the sort of *internal* propositions. When we quantify over predicative propositions using Π , we obtain a proposition (or *external* proposition). By dividing propositions into those of sort $*$ and those of sort $Prop$, we prevent the ability (and hence the need) to instantiate propositional variables with propositions constructed using Π . This is the central reason why the proof checker is *not* a proof checker for higher-order logic. (The proof checker is *simptcheck*, not *holcheck*.) The PTS would correspond to (intuitionistic) HOL if we identified $*$ with $Prop$.

Let's describe each of these rules. The first rule constructs function types.

- $(Type, Type, Type)$. If $A, B : Type$ is a type, then $(A \rightarrow B) : Type$ is the type of functions from A to B . We write $A \rightarrow B$ ($>$ in ASCII) instead of $\Pi x : A. B$ since B cannot depend on a variable.)

The next two rules construct propositions corresponding to universal quantification.

- $(Type, *, Prop)$. If $A : Type$ is a type and $\varphi : *$ is an internal proposition, then $(\Pi x : A. \varphi) : Prop$ is a proposition meaning $\forall x \in A. \varphi$ (written using $!$ in ASCII).
- $(Type, Prop, Prop)$. If $A : Type$ is a type and $P : Prop$ is a proposition, then $(\Pi x : A. P) : Prop$ is a proposition meaning $\forall x \in A. P$ (written using $!$ in ASCII).

The final four rules construct propositions corresponding to implication. Since the codomain cannot depend on any variable of sort $*$ or $Prop$, we write $P \rightarrow Q$ ($>$ in ASCII) instead of $\Pi u : P. Q$.

- $(*, *, Prop)$. If $\varphi, \psi : *$ are internal propositions, then $\varphi \rightarrow \psi$ is a proposition corresponding to an implication.
- $(*, Prop, Prop)$. If $\varphi : *$ is an internal proposition and $Q : Prop$ is a proposition, then $\varphi \rightarrow Q$ is a proposition corresponding to an implication.
- $(Prop, *, Prop)$. If $P : Prop$ is a proposition and $\psi : *$ is an internal proposition, then $P \rightarrow \psi$ is a proposition corresponding to an implication.
- $(Prop, Prop, Prop)$. If $P, Q : Prop$ are propositions, then $P \rightarrow Q$ is a proposition corresponding to an implication.

The type system of *simptcheck* is strong enough to allow one to write a signature specifying first-order logic and axiomatic set theory. Hence the type system is strong enough for one to specify mathematical objects, propositions, and proofs.

A Signature for First Order Set Theory

In this chapter we give a signature for first-order logic and ZFC set theory. This signature is closely related to the dependently typed set theory signature described in detail in [5]. This is included with the distribution of *simptcheck*. It is the first part of the file `examples/zfc-1-claims`. To check this file, you can do

```
simptcheck examples/zfc-1-claims
```

or (for example)

```
simptcheck -V -r examples/zfc-1-claims
```

to get some output. Many claims are declared in `examples/zfc-1-claims`. To check the proofs of these claims, and to ensure that all claims are proven, you can do

```
simptcheck -r examples/zfc-1-claims -s examples/zfc-1-proofs
```

Warning: This may take a while. On my machine it takes about 5 and a half minutes.

Usually you only want to check the proofs of the specific claims you're interested in, and accept all other claims. Presumably you've checked proofs for these other claims in the past.

1. First-Order Logic with Equality

We assume a constant $V : Type$ of sort $Type$, which will be the universe of set theory. We need not explicitly declare V . Next we declare constants corresponding to equality, negation, implication, equivalence, conjunction, disjunction, and quantifiers over V .

```
const eq:V>V>*;
const not:*>*;
const imp:*>*>*;
const equiv:*>*>*;
const and:*>*>*;
const or:*>*>*;
const all:(V>*)>*;
const ex:(V>*)>*;
const exu:(V>*)>*;
```

Next we declare constants corresponding to natural deduction rules for equality, negation, implication, equivalence, and universal quantifiers.

```
known eqE:!x:V.!y:V.!phi:V>*.eq x y>phi x>phi y;
known xmcases:!phi:*.!psi:*. (phi>psi)>(not phi>psi)>psi;
known notE:!phi:*.!psi:*.phi>not phi>psi;
known impI:!phi:*.!psi:*. (phi>psi)>imp phi psi;
```

```

known impE:!phi:*.!psi:*.imp phi psi>phi>psi;
known equivI1:!phi:*.!psi:*.phi>psi>equiv phi psi;
known equivI2:!phi:*.!psi:*.not phi>not psi>equiv phi psi;
known equivEimp1:!phi:*.!psi:*.equiv phi psi>phi>psi;
known equivEimp2:!phi:*.!psi:*.equiv phi psi>psi>phi;
known allI:!phi:V>*.(!x:V.phi x)>all (\x.phi x);
known allE:!phi:V>*.all (\x.phi x)>!x:V.phi x;

```

We assume axioms corresponding to definitions for conjunction, disjunction, existential quantification, and the quantifier for unique existence.

```

known andEquiv:!phi:*.!psi:*.equiv (and phi psi)
                                (not (imp phi (not psi)));
known orEquiv:!phi:*.!psi:*.equiv (or phi psi) (imp (not phi) psi);
known exEquiv:!phi:V>*.equiv (ex (\x.phi x))
                                (not (all (\x.not (phi x))));
known exuEquiv:!phi:V>*.equiv (exu (\x.phi x))
                                (ex (\x.and (phi x) (all (\y.imp (phi y) (eq x y)))));

```

2. A Description Operator

It is useful to also include a description operator extracting an element x (of type V) from a property p (of type $V \rightarrow *$) if x is the only element satisfying V .

```

const descr:(V>*)>V;
known descrp:!phi:V>*.exu (\x.phi x)>phi (descr (\x.phi x));

```

3. Set Theory

We declare a constant in corresponding to the membership relation \in .

```

const in:V>V>*;

```

Next we declare constants corresponding to set constructors and axioms. First extensionality:

```

known setextAx:all (\A.all (\B.imp (all (\x.equiv (in x A) (in x B)))
                                (eq A B)));

```

Next the empty set:

```

const emptyset:V;
known emptysetAx:all (\x.not (in x emptyset));

```

Instead of taking unordered pairs as primitive, we take set adjunction (taking x and A to $\{x\} \cup A$) as primitive.

```

const setadjoin:V>V>V;
known setadjoinAx:all (\x.all (\A.all (\y.equiv
                                (in y (setadjoin x A)) (or (eq y x) (in y A)))));

```

The power set:

```

const powerset:V>V;
known powersetAx:all (\A.all (\B.equiv (in B (powerset A))
                                (all (\x.imp (in x B) (in x A)))));

```

The union of a set of sets gives a set:

```

const setunion:V>V;
known setunionAx:all (\A.all (\x.equiv (in x (setunion A))
                                (ex (\B.and (in x B) (in B A))))));

```

There is an infinite set ω which is the least set containing the emptyset and closed under the act of adjoining a member of ω with itself (i.e., the ordinal successor function).

```

const omega:V;
known omega0Ax:in emptyset omega;
known omegaSAx:all (\x.imp (in x omega) (in (setadjoin x x) omega));
known omegaIndAx:all (\A.imp (and (in emptyset A)
                                (all (\x.imp (and (in x omega) (in x A)) (in (setadjoin x x) A))))
                                (all (\x.imp (in x omega) (in x A))));

```

The replacement axiom:

```

known replAx:!phi:V>V>*.all (\A.imp (all
                                (\x.imp (in x A) (exu (\y.phi x y))))
                                (ex (\B.all (\x.equiv (in x B) (ex (\y.and (in y A) (phi y x)))))));

```

The foundation axiom:

```

known foundationAx:all (\A.imp (ex (\x.in x A)) (ex (\B.and (in B A)
                                (not (ex (\x.and (in x B) (in x A)))))));

```

A version of the well-ordering principle (which is equivalent to the axiom of choice):

```

known wellorderingAx:all (\A.ex (\B.and (and (and
                                (all (\C.imp (in C B) (all (\x.imp (in x C) (in x A))))
                                (all (\x.all (\y.imp (and (in x A) (in y A))
                                        (imp (all (\C.imp (in C B) (equiv (in x C) (in y C))) (eq x y))))))
                                (all (\C.all (\D.imp (and (in C B) (in D B)) (or
                                        (all (\x.imp (in x C) (in x D))
                                        (all (\x.imp (in x D) (in x C))))))))
                                (all (\C.imp (and (all (\x.imp (in x C) (in x A)) (ex (\x.in x C))
                                        (ex (\D.ex (\x.and (and (and (in D B) (in x C))
                                                (not (ex (\y.and (in y D) (in y C))))
                                                (all (\E.imp (in E B)
                                                        (or (all (\y.imp (in y E) (in y D))
                                                                (in x E))))))))))))));

```

Finally, though it is redundant, we include a set constructor for forming sets $\{x \in A \mid \varphi(x)\}$ given A and φ .

```

const dsetconstr:V>(V>*)>V;
known dsetconstrI:!A:V.!phi:V>*.!a:V.in a A>phi a>
                                in a (dsetconstr A (\b.phi b));
known dsetconstrEL:!A:V.!phi:V>*.!x:V.in x (dsetconstr A (\a.phi a))>
                                in x A;
known dsetconstrERa:!A:V.!phi:V>*.!a:V.in a A>
                                in a (dsetconstr A (\b.phi b))>phi a;

```


Organization of the Code

In this chapter we give an overview of the code. For more detail, one can look at the `mli` interface files. Since the code is written in OCaml, it is relatively readable.

The main file is `simptcheck.ml`. The code in this file reads the command line arguments, checks the files indicated, and prints corresponding information. The program exits with 0 if there is no error or type checking problem. Otherwise the program exits with 1.

The code depends on the parsing code, the code for representing syntax, code for handling the state, and the actual type checking/proof checking code.

1. Lexer and Parser

The parsing code is generated using `ocamllex` and `ocamlyacc`. The file `ocamllex` uses to generate the lexer is `lexer.mll`. The file `ocamlyacc` uses to generate the parser is `parser.mly`. The parsing code uses the datatypes for preterms and document items defined in `parsedata.ml` (with interface `parsedata.mli`). All types, terms, props and proofs are parsed as *preterms*. The distinction between the four kinds of preterms is made in the syntax related code described next.

2. Syntax

Datatypes for types, terms, props, and proofs are defined in `syntax.ml` (with interface `syntax.mli`).

- `stp` is the type of (simple) types.
- `trm` is the type of terms.
- `prop` is the type of props.
- `pftrm` is the type of proofs.

There are also functions for converting objects of these types to strings (for printing). We use deBruijn indices (starting at 0) to represent bound variables.

3. State

As usual in functional programming, most of the code is stateless. However, we do have a certain amount of mutable state information for keeping up with the current signature, how many claims are open, etc. This information is encapsulated in an object of class `state` as defined in `state.ml` (with interface `state.mli`).

One particular aspect of the state involves ensuring that there are no circular dependencies in the signature. This can be tricky since a definition (or proof) can be given much later than the missing definition (or open claim) was given. If we did not do this kind of dependency checking, one could declare two claims and justify both in terms of the other:

```
const p:*;  
claim p1:p;  
claim p2:p;  
proof p1=p2;  
proof p2=p1;
```

Even worse, one could prove a claim by referring to itself:

```
const p:*;  
claim p1:p;  
proof p1=p1;
```

We prevent these circular dependencies by keeping a graph of signature elements where an edge indicates that one element depends on another. A definition or proof is rejected if it leads to a cycle in this graph. The code for handling these graph operations is in `digraph.ml` (with interface `digraph.mli`). We actually compute the transitive closure of the graph every time we enter a new edge. In this way, a cycle will lead to a node having an edge to itself.

4. Checker

The main code is in the file `checker.ml` (with interface `checker.mli`). This code converts preterms to types, (well-typed) terms, props or (well-propped) proofs (relative to the current state). If the preterm cannot be converted to an appropriate syntactic object, an appropriate exception will be thrown and this may be used to give an error message.

The code in `checker.ml` includes code for shifting deBruijn indices, substituting for deBruijn indices, beta normalizing, checking if two terms or propositions are equal up to $\beta\eta$ -conversion and expanding (non-abstracted) definitions. This code is vital for checking if a proof is well-formed and proves the given proposition.

Bibliography

- [1] Henk Barendregt. An introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, April 1991.
- [2] Henk Barendregt. Lambda calculi with types. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 117–309. Clarendon Press, 1992.
- [3] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.
- [4] Chad E. Brown. *Automated Reasoning in Higher-Order Logic: Set Comprehension and Extensionality in Church's Type Theory*, volume 10 of *Studies in Logic: Logic and Cognitive Systems*. College Publications, 2007.
- [5] Chad Edward Brown. *Dependently Typed Set Theory*. SEKI-Working-Paper SWP-2006-03 (ISSN 1860-5931). SEKI Publications, Saarland Univ., 2006.
- [6] J. H. Geuvers. The calculus of constructions and higher order logic. In Ph. de Groote, editor, *The Curry-Howard Isomorphism*, pages 139–191. Academia, Louvain-la-Neuve (Belgium), 1995.
- [7] Mike Gordon. Set theory, higher order logic or both? In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics. 9th International Conference, TPHOLs'96*, volume 1125 of *Lecture Notes in Computer Science*, pages 191–201, Turku, Finland, 1996. Springer-Verlag.
- [8] L. S. Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH system*. PhD thesis, Eindhoven Univ., Math. Centre, Amsterdam, 1979.
- [9] E. Landau. *Grundlagen der Analysis*. Leipzig, 1930.
- [10] Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science Publishers, 1999. In preparation.
- [11] Frank Pfenning and Carsten Schürmann. System Description: Twelf—A Meta-Logical Framework for Deductive Systems. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206, Trento, Italy, 1999. Springer-Verlag.
- [12] Freek Wiedijk. A new implementation of Automath. *J. Autom. Reasoning*, 29(3-4):365–387, 2002.