A Small Framework for Proof Checking

Hans de Nivelle and Piotr Witkowski

Institute of Computer Science, University of Wrocław, Poland
nivelle|pwit@ii.uni.wroc.pl

Abstract. We describe a framework with which first order theorem provers can be used for checking formal proofs. The main aim of the framework is to take as much advantage as possible from the strength of first order theorem provers in the formalization of realistic formal proofs. In order to obtain this, we restricted the use of higher order constructs to a minimum. In particular, we refrained from λ notation in formulas and from currying.
The first order prover can be freely chosen. All communication with the theorem prover uses TPTP syntax.
The system is intended for teaching, for checking mathematical proofs or correctness proofs of algorithms and also for improving the effectiveness of theorem provers. In its current set up, the system is not intended for building large libraries of checked mathematics.

1 Introduction

We describe a framework with which first order theorem provers can be used for checking formal proofs. The main aim of the framework is to take as much advantage as possible from the strength of first order theorem provers. In order to obtain this, we try to stay as close as possible to first order logic. The only higher order constructs in the logic are second order quantifications. Second order quantification is strong enough to express induction axioms, and set theoretic axioms.
We call the formulas that the framework uses *weak untyped second order* (WUSO) formulas. They are formally defined in Section 1.1. The system stores formulas in *contexts*. A context is essentially a stack of formulas. By specifying operators that modify contexts, the natural deduction rules $\rightarrow$-intro and $\forall$-intro can be defined. The complementary rules $\rightarrow$-elim and $\forall$-elim are obtained by explicitly specifying instances and moduls ponens combinations when a formula is used. These four rules together specify natural deduction for the $(\forall, \rightarrow)$ fragment of WUSO formulas.
All other reasoning is done by delegating reasoning tasks to a first order theorem prover. The theorem prover can be freely chosen by the user. The user can specify with which parameters the theorem prover has to be called, and how it can be recognized when the prover has found a proof.
The main aim of this work is to obtain insight into the question how useful first order theorem provers can be as assistant in the verification of realistic proofs,

and to obtain realistic test data for theorem provers. In addition, we intend to use the system as a tool for teaching logic and verification.

There have been quite a few more attempts to connect first order provers to interactive provers. (See for example [7], [2]) The main difference with these approaches is that we try to adopt the calculus as much as possible towards the theorem prover, instead of plugging the theorem prover into a calculus that is already fixed. Most interactive theorem provers use a variant of higher order logic (with currying) and a rich type system. The standard logic operators are usually defined inside the logic. Translating such formulas into first order logic is a nontrivial task, We hope that we can avoid most of the translation problems by using a logic close to the logic of the theorem prover.

In the literature, a lot of attention has been given to the problem of translating proofs found by a theorem prover back into the calculus of the interactive proof assistant. (See [8], [1], [5], [6]) Using such a translation, it can be avoided that the external theorem prover has to be trusted. In the present implementation, we completely ignore this problem. We acknowledge that this problem is important, but it is not the aspect that we want to study with the present system. We want to study the problem of the effectiveness of first order theorem proving. Our experience (from [1]) is that automated theorem provers are not as effective in solving real life problems as one would hope. Discussions with developers of interactive proof checkers confirm this experience. The problem is also mentioned in [8]. It is our hope that, by taking first order theorem proving into account from the beginning, a system can be obtained in which first order theorem proving can be more effective.

Our approach to proof checking is closely related to Mizar [10], but more basic. Mizar has a rich type system, while we don't have a type system. Mizar internally uses a very weak theorem prover, (somewhat described in [11]), which is able to do some equality reasoning, and some propositional reasoning.

In [9] a proof checking system is described that is in structure somewhat similar to our current system. Both systems use an external theorem prover for proof checking. The main difference is that we want to use our system for checking realistic proofs, while the system of [9] is intended for checking the outputs of theorem provers. In our system, if one wants to increase reliability, one can use multiple theorem provers and have each step checked by different provers.

1.1 Weak Untyped Second Order Logic

We define the fragment of *weak second order logic* used by our system. The fragment is chosen as a compromise between expressibility on one hand, and suitability for first order theorem proving on the other hand. In order to obtain sufficient expressibility, some higher-order features are necessary. In order to remain close to first order logic, we refrained from λ notation and currying in formulas. At present, the fragment is untyped, but this can easily be changed since there are no real obstacles for adding simple types to first order theorem provers.

The fragment is a *second order logic*, because it is allowed to quantify over functions and predicates that work on objects. We call the logic fragment *weak* second order logic, because second order functions cannot be used as arguments of other functions or predicates, and because λ notation is not allowed inside formulas. Although the logic is untyped, we still insist that variables are declared.

Definition 1. *A* declaration *has one of the following two forms:*

- *A function declaration FUNCTION f:n declares f as a function symbol of arity n. In case $n = 0$, the function symbol is a constant.*
- *A predicate declaration PREDICATE p:n declares p as a predicate with arity n.*

We usually abbreviate FUNCTION f:n to FUNC f:n and PREDICATE p:n to PRED p:n.

The logic is untyped, and there are no higher-order functions/predicates. Thefore, it is sufficient to specify the arity of a symbol in order to declare it.

Definition 2. *The set of* weakly untyped second order *(WUSO) formulas is recursively defined as follows:*

- *A usual (first order) atom is a WUSO formula.*
- *$\bot$ and $\top$ are WUSO formulas.*
- *If F is a WUSO formula, then $\neg F$ is a WUSO formula.*
- *If F_1 and F_2 are WUSO formulas, then $F_1 \wedge F_2$, $F_1 \vee F_2$, $F_1 \to F_2$, and $F_1 \leftrightarrow F_2$ are also WUSO formulas.*
- *If F is a WUSO formula, $D_1, \ldots, D_n$, $n > 0$, is a sequence of declarations, then $\forall D_1, \ldots, D_n\ F$ and $\exists D_1, \ldots, D_n\ F$ are WUSO formulas.*

Quantifications of form $\forall\exists$ FUNC x_1:$0, \ldots, x_n$:$0\ F$ are called first order*. We usually abbreviate first order quantifications to $\forall\exists\ x_1 \cdots x_n\ F$.*
A WUSO formula is called schematic first order *if it has form $\forall D_1, \ldots, D_n\ F$ or form F, and F contains only first order quantifications.*

In addition to satisfying Definition 2, a formula must be *well formed*, which means that all symbols occurring in it have to be declared.
Our fragment is a bit stronger than the logic used by Mizar [10], which uses schematic first order formulas, but we will probably not make use of this fact in applications. We now give some examples of WUSO formulas:

Example 1. The induction schema for natural numbers:

$$\forall \text{ PRED } p\text{:}1\ p(0) \wedge [\ \forall \text{ FUNC } n\text{:}0\ N(n) \to P(n) \to P(\text{succ}(n))]$$

$$\to \forall \text{ FUNC } m\text{:}0\ N(m) \to P(m).$$

The axiom of separation (Aussonderungsaxiom):

$$\forall \text{ PRED } p\text{:}1\ \forall \text{ FUNC } x\text{:}0\ \exists \text{ FUNC } y\text{:}0$$

$$(\forall \text{ FUNC } \alpha\text{:}0\ \alpha \in y \leftrightarrow \alpha \in x \wedge p(\alpha)).$$

1.2 Contexts

The system collects all its assumptions, declarations and proven theorems in a context.

Definition 3. *A* context *Γ is a sequence of form $\Gamma_1, \ldots, \Gamma_p$, $p \geq 0$. Each Γ_i either has form C_i or form $L_i{:}C_i$. Each C_i in turn must have one of the forms listed below. Each L_i, when it is present, is a label. We explain in Definition 4 under which conditions C_i can have a label. Here we list the possible forms of the C_i.*

- *A declaration of form FUNC $f{:}n$ or PRED $p{:}n$.*
- *A definition of form FUNC $f := \lambda x_1 \cdots x_n\ t$, or of form PRED $p := \lambda x_1 \cdots x_n\ F$.*
- *An indirect function definition of form FUNC $f := \lambda x_1 \cdots x_n\ y\ F$. The other definitions are called* direct.
- *An assumption F.*
- *A proven formula F.*

In the list, F denotes a WUSO formula, t a first order term.

Note that λ abstraction cannot be used in formulas, only in definitions and in substitutions. Because abstraction is possible only on 0-arity function variables, there is no need to include type information in an abstraction.
An indirect function definition defines an n-ary function through an $(n+1)$-ary predicate. In order to be accepted by the system, the user has to provide proofs of $\forall x_1 \cdots x_n \exists y\ F(x_1, \ldots, x_n, y)$ and of $\forall x_1 \cdots x_n \forall y_1 y_2\ F(x_1, \ldots, x_n, y_1) \wedge F(x_1, \ldots, x_n, y_2) \rightarrow y_1 = y_2$.

Definition 4. *Most of the elements C_i that can occur in a context have a meaning that can be expressed by a formula. We call this formula* the characteristic formula *of C_i. It is defined as follows:*

- *A declaration of form FUNC $f{:}n$ or* PRED *$p{:}n$ has no characteristic formula.*
- *If C_i has form FUNC $f := \lambda x_1 \cdots x_n\ t$, then the characteristic formula equals*

$$\forall x_1 \cdots x_n\ f(x_1, \ldots, x_n) = t[x_1, \ldots, x_n].$$

  The characteristic formula of PRED $p := \lambda x_1 \cdots x_n\ F$ equals

$$\forall x_1 \cdots x_n\ p(x_1, \ldots, x_n) \leftrightarrow F[x_1, \ldots, x_n].$$

- *The characteristic formula of an indirect function definition FUNC $f := \lambda x_1 \cdots x_n\ y\ F$ equals*

$$\forall x_1 \cdots x_n\ y\ f(x_1, \ldots, x_n) = y \leftrightarrow F[x_1, \ldots, x_n, y].$$

- *The characteristic formula of a formula assumption F equals F.*
- *The characteristic formula of a proven formula F equals F.*

A context element C_i can have a label exactly when it has a characteristic formula. The purpose of the label is to assign a name to the characteristic formula.

1.3 Forward Reasoning

The calculus has three mechanisms for forward reasoning. These are instantiation, modus ponens and first order reasoning. There is also a mechanism for conditional reasoning, which will be discussed in the next section. The reasoning mechanisms (already without first order theorem proving) cover the usual natural deduction rules for $\forall$ and $\rightarrow$.
Instantiation is the following rule: From $\forall x\ F$ derive $F[x := t]$. Modus ponens is the rule: From A and $A \rightarrow B$ derive B. Instantiation and modus ponens are handled together in *references*. References are used in first order reasoning steps for referring back to formulas that have been proven before. In the references, one can specify which instantiations have to be used, and how modus ponens must be applied.
First-order reasoning is delegated to a first order theorem prover, which can be chosen by the user. Every time a first order reasoning step has to be made, the system prepares an input file in TPTP-syntax, starts the theorem prover, waits for a result, and checks the outputfile for a characteristic string that indicates that a proof was found. At this moment, we do not attempt to check the proof that was found by the theorem prover. The system is designed such a way that it is possible to run each goal on more than one theorem prover, in case one wants to avoid trusting a single theorem prover.
For each first order reasoning step, the user has to indicate its result, and he has to indicate from which premisses he expects the result to be provable. If the proof succeeds, the new formula will be added to the context as a proven formula. The user can assign a label to the formula. The general schema is given in Section 2.1. Although it is a bit more work for the user, listing the premisses avoids that the theorem prover has to be called with large input sets.
In order to prove a new formula using a context Γ, every characteristic formula of an element of Γ can be used. In order to refer to the characteristic formulas, one can make use of the labels. Additionally, indirect references of form '3 formulas after label X' , '2 formulas before label X', 'the last formula', or 'the second last formula' are allowed.

Definition 5. *Given a context Γ, we recursively define the set of references and the formulas that they refer to:*

- *A label L is a reference. In case Γ contains an element with label L, the reference refers to the characteristic formula of L.*
- *An expression of form $L + i$ or $L - i$ is a reference. In case Γ contains an element with label L, the reference refers to the i-th characteristic formula after (or before) L. The references $L+0$ and $L-0$ refer to the same formula as L.*
- *An expression of form $-i$ is a reference. In this case $-i$ refers to the i-th characteristic formula from the end of Γ. The last characteristic formula in Γ can be referred to by -1.*
- *If R is a reference, then $R\Theta$ is a reference. Θ must a be a substitution of form*

$$\{\ FUNC\ f_1 := \lambda \overline{x}_1 t_1, \ldots,\ FUNC\ f_m := \lambda \overline{x}_m t_m,$$

$$PRED\ p_1 := \lambda \overline{y}_1 F_1, \ldots,\ PRED\ p_n := \lambda \overline{y}_n F_n\ \}.$$

If R refers to a formula of form

$$\forall\ FUNC\ f_1 \cdots\ FUNC\ f_m\ \ PRED\ p_1 \cdots\ PRED\ p_n\ F,$$

(possibly after reshuffling top level $\forall$ quantifiers), then $R\Theta$ refers to $F\Theta$.

– *If R_1 and R_2 are references, then $MP(R_1, R_2)$ is also a reference. If R_1 refers to a formula A, and R_2 refers to a formula of form $A \to B$, then $MP(R_1, R_2)$ refers to formula B.*

The reader may think that MP is superfluous because it is a first order rule. The reason that we added it separately is the fact that, although MP is a first order rule, it can work on formulas that are not first order. When there is no ambiguity, we will omit the type indicators FUNC and PRED in substitutions.

Example 2. In Example 1, the induction scheme can be instantiated by $\{p := \lambda x\ x + 0 = x\}$. The result is

$$0 + 0 = 0 \wedge [\ \forall\ \text{FUNC}\ n{:}0\ N(n) \to n + 0 = n \to \text{succ}(n) + 0 = \text{succ}(n)\ ] \to$$

$$\forall\ \text{FUNC}\ m{:}0\ N(m) \to m + 0 = m.$$

The separation axiom can be instantiated by $\{\ p := \lambda x\ \text{interesting}(x)\ \}$. The result is

$$\forall\ \text{FUNC}\ x{:}0\ \exists\ \text{FUNC}\ y{:}0$$

$$\forall\ \text{FUNC}\ \alpha{:}0\quad \alpha \in y \leftrightarrow \alpha \in x \wedge \text{interesting}(\alpha).$$

It is also possible to instantiate with $\{\ p := \lambda x\ \ \neg\text{interesting}(x),\ x := \lambda\ \text{nat}\ \}$. The result is

$$\exists\ \text{FUNC}\ y{:}0\ \forall\ \text{FUNC}\ \alpha{:}0\quad \alpha \in y \leftrightarrow \alpha \in \text{nat} \wedge \neg\text{interesting}(\alpha).$$

(The last set y can be proven empty by a simple induction argument)

1.4 Conditional Reasoning

Conditional reasoning handles the introduction and dropping of assumptions, and the introduction and dropping of eigenvariables. When an assumption A is dropped, every formula F that was proven in the context of A, has to be replaced by $A \to F$. When an eigenvariable x is dropped, every formula F that is proven in the context of x, has to be replaced by $\forall x\ A$.
In our system, conditional reasoning is handled by modifications on the context Γ. Suppose that Γ has form Γ_1, C, Γ_2, and that we want to drop C. We specify for each element in Γ_2, how it will be modified. It is not always possible to define a meaningful effect on each element of Γ_2, but we try to be as general as possible. When for some element of Γ_2, no meaningful effect can be defined, it is forbidden to drop C.

– If C is a declaration of form FUNC $c{:}0$, then Γ_2 must consist of definitions and proven formulas only. C is removed by the following procedure: As long as C is not the last element of Γ_2, the complete context can be written in the form Γ_1, C, D, Δ. We will exchange C with D. During the replacement, D is modified, and possibly also Δ. The exchanges are repeated until C reaches the end of Γ_2. Then it can be removed without consequences.
  Write Γ_2 in the form C, D, Δ, and assume that D is a definition with form FUNC $f := \lambda x_1 \cdots x_n\ t$.
  Then D is replaced by FUNC $f' := \lambda c\ x_1 \cdots x_n\ \ t$, and Δ is replaced by $\Delta\{\ f := \lambda x_1 \cdots x_n\ f'(c, x_1, \ldots, x_n)\ \}$.
  If D has form PRED $p := \lambda x_1 \cdots x_n\ F$ then it is treated analogeously. D is replaced by PRED $p' := \lambda c\ x_1 \cdots x_n\ \ F$, and Δ is replaced by $\Delta\{\ p := \lambda x_1 \cdots x_n\ p'(c, x_1, \ldots, x_n)\ \}$.
  Indirect function definitions are dealt with in the same way. We omit the details.
  If D is a proven formula F, then it is replaced by $\forall$FUNC $c{:}0\ \ F$, and Δ is not changed.
– If C is a declaration of form FUNC $f{:}n$ with $n \neq 0$, or of form PRED $p{:}n$, then Γ_2 must consist only of proven formulas. Each proven formula F is replaced by $\forall$ FUNC $f{:}n\ F$. (or by $\forall$ PRED $p{:}n\ F$)
– If C is a direct function definition of form FUNC $f{:}n := \lambda x_1 \cdots x_n\ t$, then Γ_2 is replaced by $\Gamma_2\ \{\ f := \lambda x_1 \cdots x_n\ \ t\ \}$.
  Direct predicate definitions are substituted away in the same way.
– If C is an indirect function definition, it cannot be dropped, because we have no way of substituting it away.
– If C is a formula assumption of form F, then Γ_2 must consist of proven formulas $F_1, \ldots, F_n$ only. Each formula F_i is replaced by $F \to F_i$.

We think that most of the modifications on Γ_2 are more or less obvious, except for the first case, where a 0-arity function variable is dropped. We give an example of this situation:

Example 3. Consider the context

FUNC $n{:}0$,
PRED $E := N(n) \wedge \exists$ FUNC $m{:}0\ \ N(m) \wedge m + m = n$,
PROVEN $E \to \exists$ FUNC $m{:}0\ \ d(m) = n$.

The propositional variable E means 'n is even', $N(n)$ denotes 'n is a natural number', and d denotes the doubling function $\lambda x\ x + x$.
Suppose that we want to drop the first assumption FUNC $n{:}0$. Then the definition and the proven formula have to be modified. First, the definition PRED $E := N(n) \wedge \exists$ FUNC $m{:}0\ \ N(m) \wedge m + m = n$ is replaced by PRED $E' := \lambda n\ N(n) \wedge \exists$ FUNC $m{:}0\ N(m) \wedge m + m = n$, and in the proven formula, the substitution $\{\ E := E'(n)\ \}$ is made.
After that, the formula $E'(n) \to \exists$ FUNC $m{:}0\ \ d(m) = n$ is replaced by
$\forall$ FUNC $n{:}0\ \ E'(n) \to \exists$ FUNC $m{:}0\ \ d(m) = n$.

The resulting context is

$$\text{PRED } E' := \lambda n\ N(n) \wedge \exists \text{ FUNC } m{:}0\ \ N(m) \wedge m + m = n,$$
$$\forall \text{ FUNC } n{:}0\ \ E'(n) \rightarrow \exists \text{ FUNC } m{:}0\ d(m) = n.$$

A practical implementation will try to reuse the identifier E, instead of replacing E by E'. Note that if one would use the Curry-Howard isomorphism, the two types of modifications, (adding a parameter to a definition, and adding a universal quantifier to a proven formula) would be the same, because under the Curry-Howard isomorphism, definitions and proofs of theorems are the same.

2 Proof Structure

The input to the system consists of a file containing the proof. The system is a batch system. It reads the proof, checks the steps in it, and reports errors. While reading the proof, the system maintains a context Γ, which is updated after every proof step. We list some of the constructions that can occur in proofs. The FROM-rule handles the forward reasoning by the external theorem prover. Most of the other reasoning rules are straightforwardly based on the context modifications that we defined in Section 1.4.

2.1 From

`FROM` is the rule for first order forward reasoning, it is analogeous to the `by` rule of Mizar. It has form:

$$\text{PROVE } L{:}F \text{ FROM } R_1, \ldots, R_n.$$

The $R_1, \ldots, R_n$ must be references that refer to a first order formula. F must be a first order formula. The system calls the external theorem prover which tries to prove F from the formulas denoted by the first order references $R_1, \ldots, R_n$. If it succeeds, F is added to the context as a proven formula. The label L is optional. If a label is present, F will receive label L.

2.2 Permanent Predicate/Function Definitions

A function or predicate definition has one of the following three forms:

$$\text{DEFINE FUNC } D \text{ INDIRECTLY BY } L : E$$

$$\text{EXISTENCE } R_1, \ldots, R_m \text{ UNIQUENESS } S_1, \ldots, S_n.$$

$$\text{DEFINE FUNC } D \text{ BY } L : E, \text{ or}$$

$$\text{DEFINE PRED } D \text{ BY } L : E.$$

D is the identifier being defined. L is a optional label, that will be used for the characteristic formula. E is an expression of form $\lambda x_1 \cdots x_n\ y\ F$, in which F is a formula or a term, dependent on the type of the definition.
In case of an indirect definition, $R_1, \ldots, R_m$ is a list of references from which the theorem prover must be able to prove

$$\forall x_1 \cdots x_n \exists y\ F[x_1, \ldots, x_n, y].$$

$S_1, \ldots, S_n$ is a list of references from which the theorem prover must be able to prove

$$\forall x_1 \cdots x_n \forall y_1 y_2\ F[x_1, \ldots, x_n, y_1] \wedge F[x_1, \ldots, x_n, y_2] \rightarrow y_1 = y_2.$$

2.3 Local Assumptions

A *local assumption block* has form

$$\text{ASSUME } D_1, \ldots, D_n \text{ IN } P_1, \ldots, P_m \text{ END .}$$

Each D_i has one of the following five forms:

1. PREDICATE $p{:}n$,
2. PREDICATE $p := \lambda x_1 \cdots x_n\ F$,
3. FUNCTION $f{:}n$,
4. FUNCTION $f := \lambda x_1 \cdots x_n\ t$,
5. FORMULA F, in which F is a WUSO formula.

The sequence $P_1, \ldots, P_m$ must be a proof by itself. The system first adds the assumptions $D_1, \ldots, D_n$ to the context. After that, it reads the proof $P_1, \ldots, P_m$, which can make further additions to the context. When reading of $P_1, \ldots, P_m$ is complete, the assumptions $D_1, \ldots, D_n$ are dropped from the context in the order $D_n, D_{n-1}, \ldots, D_1$. The additions, made by the proof $P_1, \ldots, P_m$, are modified according to the rules of Section 1.4.

2.4 Permanent Assumptions

A permanent assumption block has form

$$\text{ASSUME } D_1, \ldots, D_n.$$

Each D_i must have one of the following three forms:

1. PREDICATE $p{:}n$,
2. FUNCTION $f{:}n$,
3. FORMULA F.

3 Conclusions and Future Work

The system is only intended as a first attempt. Probably the most important modification that has to be made, is to add a simple type system. Simple types are very easy to implement in resolution or tableaux. Unfortunately, still none of the existing theorem provers supports simple types. We will extend the next version of Geo with simple types. We also plan to redo the verifications of [3] and of [4] in our system.
The system can be obtained from the homepage of the second author. If the system turns out succesful enough, and stabilizes, we will rewrite it with a trusted code base.

References

1. Marc Bezem, Dimitri Hendriks, and Hans de Nivelle. Automated proof construction in type theory using resolution. *Journal of Automated Reasoning*, 29(3-4):253–275, December 2002.
2. Jean-François Couchot and Stéphane Lescuyer. Handling polymorphism in automated deduction. In Frank Pfenning, editor, *Automated Deduction - CADE-21*, volume 4603 of *LNAI*, pages 263–278. Springer Verlag, 2007.
3. Hans de Nivelle and Ruzica Piskac. Verification of an off-line checker for priority-queues. In Peter H. Schmitt, editor, *Proceedings of the 3d IEEE International Conference on Software Engineering and Formal Methods*, pages 210–219, Koblenz, September 2005. IEEE Computer Society Press.
4. Hans de Nivelle and various authors. Verification of the unification algorithm. `www.ii.uni.wroc.pl/~nivelle/teaching/interactive2007/index.html`.
5. Joe Hurd. Integrating gandalf and hol. In *Theorem Proving in Higher Order Logics*, volume 1690 of *LNCS*, pages 311–321. Springer Verlag, 1999.
6. Andreas Meier. TRAMP: Transformation of machine-found proofs into natural deduction proofs at the assertion level. In D. McAllester, editor, *Proceedings of the 17th Conference on Automated Deduction (CADE–17)*, volume 1831 of *LNAI*, pages 460–464, Pittsburgh, USA, 2000. Springer Verlag, Berlin, Germany.
7. Jia Meng and Larry Paulson. Translating higher-order problems to first-order clauses. In Geoff Sutcliffe, Renate Schmidt, and Stephan Schulz, editors, *ESCoR (CEUR Workshop Proceedings)*, volume 192, pages 70–80, 2006.
8. Jia Meng, Claire Quigley, and Lawrence Paulson. Automation for interactive proof: First prototype. *Information and Computation*, 204(10):1575–1596, 2006.
9. Geoff Sutcliffe. Semantic derivation verification: Techniques and implementation. *International Journal on Artificial Intelligence Tools*, 15(6):1053–1070, 2006.
10. Freek Wiedijk. Writing a Mizar article in nine easy steps. can be obtained from homepage of author.
11. Freek Wiedijk and Andrzej Trybulec. Checker. `http://www.cs.ru.nl/~freek/` .