Common Syntax of the DFG-Schwerpunktprogramm "Deduktion"

Reiner Hähnle
Fakultät für Informatik
Universität Karlsruhe
D-76128 Karlsruhe
reiner@ira.uka.de

Manfred Kerber
School of Computer Science
The University of Birmingham
Birmingham, B15 2TT, England
M.Kerber@cs.bham.ac.uk

Christoph Weidenbach
Max-Planck-Institut für Informatik
Im Stadtwald
66123 Saarbrücken
weidenb@mpi-sb.mpg.de

Abstract

A common exchange format for logic problems to be used by members of the DFG-Schwerpunktprogramm "Deduktion" is introduced. It is thought to be an internal format that can easily be parsed such that it forms a compromise between the needs of the different groups. It is not intended to be a high-level general logic language that is easy to read or to write. The language is more general than other popular exchange formats such as Otter or TPTP in allowing non-clausal and sorted formulas as well as user-defined operators and quantifiers. The latter feature makes it also useful for non-classical logics.

1 Introduction

The language proposed in the following is intended to be a common exchange format for logic problem settings. It is thought to be an internal format that can easily be parsed such that it forms a compromise between the needs of the different groups. Therefore, it is kept *as simple as possible.* This language is not intended to be a high-level general logical language that is easy to read or to write.

In any case it will be necessary to provide tools that transform files from the present syntax into other standard formats (e.g., Otter [6] or TPTP [9]) and vice versa.

Terminal symbols are underlined or one of the following symbols ')', '(', '.', ',', or '-'.

2 Problems

What we exchange are problems. Problems are stand alone files, with respect to possible inclusions (see Section 6), that describe one specific first-order logic problem.

```
problem  ::=  begin_problem(label).
              description
              logical_part
              {settings}*
              end_problem.
```

The `description` provides at least information for the identification of the problem like problem's name, author's name, version, date, and informal description (see Section 4). If the logical part contains non-standard operators or quantifiers they have to be explained here. The explanation should contain, if possible, translation rules to standard first-order logic. Settings contain problem specific setting information as well as prover dependent strategic directives like restrictions on the term depth, directives for the compiler, set-of-support etc. The logical_part contains the pure problem. The problem should be uniquely determined by this part alone.

3 The Logical Parts

Identifiers for function, predicate and sort symbols, constants are subject to the unique name assumption and mutually exclusive with the set of terminal symbols. The same holds for variables within the same scope of quantifiers.

```
logical_part  ::=  {symbol_list} {declaration_list} {formula_list}*
```

All signature symbols as well as additional operators and quantifiers have to be declared in advance in the `symbol_list` of a problem. An arity may be given in the declaration. The arity may either be "-1", meaning arbitrary arity, or a non-negative number for some fixed arity. If no arity is given, the symbol is declared with fixed arity according to the formula part. Therefore, symbols with varying arity have to be declared explicitely.

```
symbol_list  ::=  list_of_symbols.
                  {functions[func_sym {arity} {,func_sym {arity} }*].}
                  {predicates[pred_sym {arity} {,pred_sym {arity}}*].}
                  {sorts[sort_sym {,sort_sym}*].}
                  {operators[op_sym {arity} {,op_sym {arity}}*].}
                  {quantifiers[quant_sym {arity} {,quant_sym {arity}}*].}
                  end_of_list.
      arity  ::=  -1 | number
```

The sort declarations are optional. They can be mapped to standard first-order logic by the usual relativization rules. Sorts may be used as unary predicate symbols in the logical part.

```
declaration_list  ::=  list_of_declarations.
                       {declaration}*
                       end_of_list.
     declaration  ::=  subsort_decl | term_decl | pred_decl
    subsort_decl  ::=  subsort(sort_sym,sort_sym).
        sort_sym  ::=  identifier
       term_decl  ::=  forall(var_list,term_expr) | term_expr.
       term_expr  ::=  sort_sym(term)
       pred_decl  ::=  predicate(pred_sym{,sort_sym}+).
```

We treat axioms and conjectures differently. Strategies like goal-oriented reasoning are specified in the settings part. Optionally each formula can be given a label. The list of formulas is logically interpreted to the conjunction of its parts. Logically, the axioms list and the conjectures list form an implication: *axioms* $\supset$ *conjectures*. Therefore, the formulas to be proved should not be put in negated form in the conjecture list.

```
formula_list  ::=  list_of_axioms | list_of_conjectures.
                   {formula({formula | cnf_clause | dnf_clause}{,label}).}*
                   end_of_list.
       label  ::=  pred_expr
```

Formulas are either quantified expressions or ground expressions.

```
   formula  ::=  quant_formula | log_op_formula | atom
cnf_clause  ::=  forall(var_list,cnf_clause_body) | cnf_clause_body
dnf_clause  ::=  exists(var_list,dnf_clause_body) | dnf_clause_body
```

Clauses are a subset of the set of formulas. We do *not* allow implicit quantification, that is, formulas and clauses must not contain free variables. All unbound simple terms are interpreted as constants. The rationale behind this is that the interpretation of a problem should be determined by the problem format, not by the prover. In Otter, e.g., free variables in clauses are implicitly universally quantified, while variables in formulas are implicitly existentially quantified. Moreover, variables must begin with `u,v,w,...`, constants with `a,b,c,...`. We find this confusing.

```
cnf_clause_body  ::=  or(literal{,literal}*)
dnf_clause_body  ::=  and(literal{,literal}*)
        literal  ::=  un_op(atom) | atom
```

Note that unit clauses are of the form 'or(literal)'. The empty clause may be specified as 'or(false)'.

```
quant_formula  ::=  quantifier(term_list,formula) | quantifier(var_list,formula)
   quantifier  ::=  forall | exists | quant_sym
    quant_sym  ::=  identifier
```

For single variables the notation is still: 'forall([x],phi)'. Quantifiers different from `forall` and `exists` have to be declared in advance. For the standard first-order logic quantifiers we only allow variables in the `term_list` or `var_list`. However, we can think of other quantifiers, e.g. a modal believe operator indexed with some agent, where a "quantification" on other terms makes sense. In addition to the `term_list`, the `var_list` allows the introduction of sorts.

```
     var_list  ::=  [variable_expr{,variable_expr}*]
    term_list  ::=  [term{,term}*]
variable_expr  ::=  variable_sym | sort_sym(variable_sym)
 variable_sym  ::=  identifier
```

Formulas that do not start with a quantifier are built in the usual way.

```
log_op_formula  ::=  un_op(formula) | bin_op(formula,formula) |
                     n_op(formula{,formula}*)
         un_op  ::=  not | op_sym
        bin_op  ::=  implies | implied | equiv | op_sym
          n_op  ::=  and | or | op_sym
```

The operators `or` and `and` are available for all arities greater than 0.

```
   atom  ::=  zero_op | pred_expr | equation
zero_op  ::=  true | false | op_sym
 op_sym  ::=  identifier
```

We think of 0-ary connectives (truth values) as atoms.

```
pred_expr  ::=  pred_sym{(term{,term}*)} | sort_sym(term)
 equation  ::=  equal(term,term)
 pred_sym  ::=  identifier
```

0-ary predicates are allowed.

```
          term  ::=  constant | variable | fun_sym(term{,term}*)
      constant  ::=  identifier
       fun_sym  ::=  identifier

    identifier  ::=  {letter | digit | special_symbol}+
        letter  ::=  a-z | A-Z

        number  ::=  {digit}+
         digit  ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
special_symbol  ::=  _
```

3.1 Example without Sorts

We start with a complete description of Pelletier's [7] problem No. 57:

```
begin_problem(Pelletier57).

list_of_descriptions.
name(Problem No. 57 from the Pelletier Collection).
author(F.J. Pelletier, ``Seventy-Five Problems for Testing Automatic
Theorem Provers'', Journal of Automated Reasoning, 2(2):191--216,1986).
status(unsatisfiable).
description(This is a simple problem.).
end_of_list.

list_of_symbols.
functions[f 2,a,b,c].
predicates[F].
end_of_list.

list_of_axioms.
formula(F(f(a,b),f(b,c)),ax1).
formula(F(f(b,c),f(a,c)),ax2).
formula(forall([x,y,z], impl(and(F(x,y),F(y,z)),F(x,z))),ax3).
end_of_list.

list_of_conjectures.
formula(F(f(a,b),f(a,c)),co1).
end_of_list.

end_problem.
```

3.2 Example with Sorts

We describe the logic part of an equality problem containing the natural numbers.

```
list_of_symbols.
functions(plus,s,zero).
sorts(even,nat).
end_of_list.

list_of_declarations.
subsort(even,nat).
even(zero).
forall([nat(x)],nat(s(x))).
forall([nat(x),nat(y)],nat(plus(x,y))).
forall([even(x),even(y)],even(plus(x,y))).
forall([even(x)],even(s(s(x)))).
forall([nat(y)],even(plus(y,y))).
end_of_list.

list_of_axioms.
formula(forall([nat(y)],equal(plus(y,zero),y)), ind_start).
formula(forall([nat(y),nat(z)],equal(plus(y,s(z)),s(plus(y,z)))), ind_step).
end_of_list.
```

4 Descriptions

```
description  ::=  list_of_descriptions.
                  name(text).
                  author(text).
                  {version(text).}
                  {logic(text).}
                  status(log_state).
                  description(text).
                  {date(text).}
                  end_of_list.
  log_state  ::=  satisfiable | unsatisfiable | unknown
```

We allow aribitrary `text` that is compatible with the syntax. The `logic` part is mandatory, if the problem contains non-standard operators or quantifiers.

5 Settings

The settings contain problem and system specific information such as switches, lists of formulas that are treated specially, hints for the compiler etc. The settings consist of a general setting section and various system dependent sections. The content of the general setting section is currently restricted to an enumeration of “hypotheses”, that are formulas of the conjecture part which are compatible with the axioms. The axioms in conjunction with the hypothesis are satisfiable. The general section part is open to extensions if needed.

For the system dependant sections we require a unique label for each system. There are no restrictions for the content of these sections except the compatibility with the syntax.

```
      settings ::= list_of_general_settings {setting_entry}+ end_of_list.  |
                   list_of_settings(setting_label). text end_of_list.
 setting_entry ::= hypothesis(label {,label}*)
 setting_label ::= KIV | LEM | PROTEIN | SATURATE | 3TAP | SETHEO | SPASS
```

The labels name the following systems: KIV [8], LEM [4], PROTEIN [1], SATURATE [3], $_3T^AP$ [2], SETHEO [5], SPASS [10]. For example, to specify a set of support one may (i) give labels to the formulas/clauses that have support and (ii) list these labels in a list under some keyword `set_of_support`:

```
list_of_settings(SPASS).
set_of_support(ax1,ax2,ax3).
precedence(a,b,c,f,F).
end_of_list.
```

6 Miscellaneous

6.1 Comments

After the '`%`' symbol the rest of line is ignored. Comments stretching over several lines may be enclosed by a '`/*`', $\cdots$, '`*/`' pair.

6.2 Includes

At any part of a problem, another file might be included by

`include(filename)`

Includes are a necessary means to structue problems. However, they should be used with some care because with "includes", problem files are no longer stand alone documents. In order to avoid the problem of different pathname conventions on different machines, only files being in the same directory can be included.

Acknowledgements

We would like to thank all members of the German "Schwerpunkt Deduktion" group who contributed to this paper. Special thanks to Enno Keen and Andreas Nonnengart who proof-read several preliminary versions of this paper.

References

[1] Peter Baumgartner and Ulrich Furbach. PROTEIN: A *PRO*ver with a *T*heory *E*xtension *I*nterface. In A. Bundy, editor, *12th International Conference on Automated Deduction, CADE-12*, volume 814 of *LNAI*, pages 769–773. Springer, 1994. Available in the WWW, URL: `http://www.uni-koblenz.de/ag-ki/Systems/PROTEIN/`.

[2] Bernhard Beckert, Stefan Gerberding, Reiner Hähnle, and Werner Kernig. .The Many-Valued Tableau-Based Theorem Prover $_3T^AP$. In D. Kapur, editor, *11th International Conference on Automated Deduction, CADE-11*, volume 607 of *LNAI*, pages 758–760. Springer, 1992. Available in the WWW, URL: `http://i12www.ira.uka.de/~threetap/`.

[3] Harald Ganzinger and Robert Nieuwenhuis. The Saturate System 1994. http://www.mpi-sb.mpg.de/SATURATE/Saturate.html.

[4] Birgit Heinz. *Anti-Unifikation modulo Gleichungstheorie und deren Anwendung zur Lemmagenerierung*. PhD thesis, TU Berlin, Dec 1995.

[5] Reinhold Letz, Johann Schumann, S. Bayerl, and Wolfgang Bibel. SETHEO: A High-Performance Theorem Prover. *Journal of Automated Reasoning*, 8(2):183–212, 1992.

[6] William McCune. Otter 2.0 users guide. Report ANL-90 9, Argonne National Laboratory, March 1990.

[7] Francis Jeffry Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2(2):191–216, 1986. Errata: *Journal of Automated Reasoning*, 4(2):235–236,1988.

[8] Wolfgang Reif. The KIV-approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, volume 1009 of *LNCS*. Springer, 1995.

[9] Geoff Sutcliffe, Christian Suttner, and Theodor Yemenis. The TPTP problem library. In Alan Bundy, editor, *Twelfth International Conference on Automated Deduction, CADE-12*, volume 814 of *Lecture Notes in Artificial Intelligence, LNAI*, pages 252–266, Nancy, France, June 1994. Springer.

[10] Christoph Weidenbach, Bernd Gaede, and Georg Rock. SPASS & FLOTTER, Version 0.42. Submitted, available via ftp from `ftp.mpi-sb.mpg.de` in the directory `pub/SPASS` named `fass.dvi`, see also the SPASS distribution in the very same directory `spass.0.42.tgz`, 1996.