# Designing Proof Formats

## A User's Perspective

Sascha Böhme and Tjark Weber

TECHNISCHE UNIVERSITÄT MÜNCHEN

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

First Workshop on Proof Exchange for Theorem Proving (PxTP)
Wrocław, Poland

1 August, 2011

# Why Do Proofs Matter?

Correctness is paramount: automatic provers are used, e.g., to verify safety-critical applications.

Bugs are inevitable: state-of-the-art provers are complex tools.

Verification of automatic provers may not be feasible in practice.

# Why Do Proofs Matter?

**Correctness** is paramount: automatic provers are used, e.g., to verify safety-critical applications.

**Bugs** are inevitable: state-of-the-art provers are complex tools.

**Verification** of automatic provers may not be feasible in practice.

**Certificates** for individual results are relatively easy to generate. Ideally, they can be checked independently by a simple (possibly verified) proof checker.

# Classes of Automatic Provers

- **SAT** prove unsatisfiability of CNF formulas
- **QBF** prove satisfiability and invalidity of quantified Boolean formulae
- **SMT** prove unsatisfiability of formulas from (fragments of) first-order logic with theories
- **ATP** prove validity of formulas from first-order logic with equality

# Proof Formats of Automatic Provers

## SAT

- conceptually simple: sequence of resolution steps
- no proof standard: provers have their own proof syntax

## QBF

- proofs of invalidity: based on Q-resolution
- proofs of satisfiability: diverse techniques
- proof standard proposed for competitions

# Proof Formats of Automatic Provers

## SMT

- various distinct proof formats
- based on natural deduction, LF, ...
- proof standard proposed for competitions

## ATP

- TSTP proof standard due to annual CASC
- very general: fixed syntax, flexible inferences

# LCF-style Proof Assistants

LCF-syle proof assistants are based on a small inference kernel.
Theorems are implemented as an abstract data type.

As a framework for the implementation of proof checkers, LCF-style proof assistants are ...

- generic (e.g., based on higher-order logic)
- sound (provided their kernel is correct)
- powerful (term rewriting, arithmetic, ...)

# Proof Certificates

Certificates should be:

**for provers** easy (and fast) to generate

**for users** easy and fast to check and
easy to store

# Proof Certificates

Certificates should be:

| | |
|---|---|
| for provers | easy (and fast) to generate |
| for users | easy and fast to check and |
| | easy to store |

Conflict!

# Guidelines for Proof Formats

- Use an existing format
- Provide a human-readable, lightweight representation
- Take theoretical considerations into account
- Use simple, canonical semantics
- Add declarative information
- Provide exhaustive documentation

# Guidelines for Proof Formats

- Use an existing format
- Provide a human-readable, lightweight representation
- Take theoretical considerations into account
- Use simple, canonical semantics
- Add declarative information
- Provide exhaustive documentation

# Use an Existing Format

Good:

- “Let’s add some `printf` statements.”

Much better:

- Use an existing proof format!
- Alternatively: be compatible with widespread provers.

# Guidelines for Proof Formats

- Use an existing format
- Provide a human-readable, lightweight representation
- Take theoretical considerations into account
- Use simple, canonical semantics
- Add declarative information
- Provide exhaustive documentation

# Provide a Human-Readable, Lightweight Representation

Good:

- "Let's provide an in-memory API."
- "And a binary file format."

Much better:

- Provide a human-readable representation!
- Use a standardized data format language!

# Guidelines for Proof Formats

- Use an existing format
- Provide a human-readable, lightweight representation
- Take theoretical considerations into account
- Use simple, canonical semantics
- Add declarative information
- Provide exhaustive documentation

# Take Theoretical Considerations into Account

Good:

- "Here's a function call, let's print that."
- "And this data structure too."

Much better:

- Consider complexity of proof checking!
- Proof checking ought to be easier than proof search.

# Guidelines for Proof Formats

- Use an existing format
- Provide a human-readable, lightweight representation
- Take theoretical considerations into account
- Use simple, canonical semantics
- Add declarative information
- Provide exhaustive documentation

# Use Simple, Canonical Semantics

Bad:

- "Let's use one really powerful proof rule, with numerous flags for odd cases."
- "And some rules for particular optimizations in the prover."

Much better:

- Use small, focused inference rules with clear semantics!
- Do not expose low-level optimizations!

# Guidelines for Proof Formats

- Use an existing format
- Provide a human-readable, lightweight representation
- Take theoretical considerations into account
- Use simple, canonical semantics
- Add declarative information
- Provide exhaustive documentation

# Add Declarative Information

Bad:

- Implicit invariants about formulas.
- Non-obvious assumptions.

Much better:

- Explicitly provide inferred formulas!
- Add “superfluous” information for checking!

# Guidelines for Proof Formats

- Use an existing format
- Provide a human-readable, lightweight representation
- Take theoretical considerations into account
- Use simple, canonical semantics
- Add declarative information
- Provide exhaustive documentation

# Provide Exhaustive Documentation

Bad:

- 

Much better:

- Describe the (abstract and concrete) syntax and semantics of the proof format, including preprocessing and normalization!
- Ideally provide an independent checker or some (semi-)formal documentation!

# Conclusion

- Use an existing format
- Provide a human-readable, lightweight representation
- Take theoretical considerations into account
- Use simple, canonical semantics
- Add declarative information
- Provide exhaustive documentation