Improving the ISO Prolog standard by analyzing compliance test results

Péter Szabó <pts@cs.bme.hu>
Péter Szeredi <szeredi@cs.bme.hu>
BUTE, Dept. of Computer Science and Information Theory

slides for ICLP 2006
2006-08-19 9:00–9:30
Seattle, USA

What the standard covers

The ISO Prolog Standard (ISO/IEC 13211, Part 1), published in 1995, covers:

- *syntax:* reading terms, built-in and user-defined operators
- *semantics:* term, type, unification, clause, predicate, goal, built-in control constructs (*';'/2*, *throw/1* etc.), flag, error.
- *built-in predicates:* 102 altogether, including *'='/2*, *arg/3*, *retract/1*, *stream_property/2*, *get_code/2*, *read/1*
- *expression evaluation:* what is allowed in the 2nd argument of *is/2* and how to evaluate it.

The standard includes:

- *natural language* description
- *example code* with semi-formal *expected results*
- *formal semantics:* a mathematical definition of Prolog

What is missing from the standard

Features found in most Prolog implementations, but not covered by the standard:

- *DCGs, libraries, global mutables:* standardization underway
- *blocking* calls (e.g. *freeze/2*), coroutines
- *modules:* Part 2 of the standard describes module handling, but implementations don't respect it at all
- *interactive toplevel:* line editing, displaying of results, interactive debugging
- *precompilation:* precompiling modules, dumping state
- *Prolog system startup:* command-line options, environment variables, source file search path
- *execution modes:* standard-compliant mode, legacy mode etc., startup mode selection
- *libraries handling data structures:* lists, graphs, ordered trees

Other common features missing

The following features are found in many implementations, but the differences are so too large; too early to standardize:

- *efficient string processing:* 8-bit binary strings, working with large strings, regular expressions
- *character sets:* Unicode, UTF-8, encoding and decoding during character I/O
- *CSP:* CLP(R), CLP(FD) etc.
- *external data sources:* SQL, RDF, XML etc.
- *using operating system facilities:* TCP/IP sockets, multi-threading, signals, process handling
- *GUI*
- *WWW:* HTML, HTTP, web services, XSLT etc.
- *declarations:* data types, predicate argument types, predicate modes, compile-time assertions

Scope of our work

Our long term goal is to improve portability of Prolog applications.

The goal of our current work is to *improve* (Part 1 of) the standard, and to improve standard-compliance of the implementations. A good, broadly respected standard is a good base of application portability.

We are *not extending* the standard with new features.

We have developed a *test framework* and a *test suite*, run the tests on 7 Prolog implementations, and analyzed the results.

The test suite includes the formalized variant of all the semi-formal test cases in the Prolog standard (about *680 cases*) and some additional tests we have developed (about *330 cases*).

Our tests *don't cover modules.* Test cases are slightly adjusted so that that they work no matter the implementation has a module system or not. (For example, module qualification is ignored in the 2nd arg of *clause/2*.)

Test results

system	version	#OK	#failed	#dang	#irre
SICStus	3.12.3	1010	1	0	1
aprolog	1.22	996	7	0	9
SICStus	4.0.0	986	24	0	1
gprolog	1.2.16	929	67	7	9
SWI-Prolog	5.4.7	816	158	8	30
YAP	5.0.1	632	363	7	10
Ciao-Prolog	1.10p6	541	454	7	10

Download detailed results, test cases and the test framework from `http://www.inf.bme.hu/~pts/stdprolog/`

A test case is *irre*levant if it is not applicable, e.g. test for integer overflow when integers are unbounded in the implementation. A test case is *dang*erous if running it makes the implementation eventually crash.

Why these implementations?

- Test as many *free* implementations as possible.
- Test SICStus 3 (non-free) because of historical reasons: the test suite and framework *derived* from that of SICStus 3. This also explains why SICStus 3 passes all the tests.
- Test SICStus 4 (non-free) to see how a *substantial rewrite* affects the results. Either the implementation or the test case (or both) has to be fixed for each failure.
- Test only *mature* implementations (not new or experimental): all of them appeared at least 9 years ago.
- Add aprolog to have a *minimalistic* system which was designed in standard-compliance in mind. The other implementations were born before the standard.
- We couldn't test XSB because we exceeded a limit.

We are working on a *semi-automatic* method with which implementors can add support for their implementations easily.

Test suite format


```
test_case(strprop:2, iso_doc, %8.11.8.4 % meta-info
 stream_property(S, output),                  % goal
 {[S <-- FOut ], [S <-- COut], ...}  % expectations
 [ pre((open(bar, write, FOut),           % environ-
   current_output(COut))), clean(bar)]). %     ment
```

Paragraph 8.11.8.4 of the ISO standard states that "`stream_property(S, output)` succeeds, unifying *S* with a stream-term which is open for output. On re-execution, succeeds in turn with each stream that is open for output.".
The test case above formalizes this by stating that if file `bar` (`FOut`) is opened in addition to `user_output` (`COut`), both should be returned by the call above, in any order (`{ }`), with additional possible solutions (`...`).

The *test suite* is a Prolog source file. In addition to test cases, it contains *macro definitions* (to make test case writing easier) and *small programs* (to be consulted by some test cases).

Implementation differences

Very different implementations can be standard-compliant. The most important differences are:

- *integer boundedness:* covered by unchangeable Prolog flags
- *negative integer representation*
- *maximum compound term arity* (covered by Prolog flag *max_arity*), other *limits* (such as maximum atom length) are not covered by flags.
- *execution state:* convered by changeable Prolog flags: *char_conversion*, *debug*, *unknown*, *double_quotes*.
- *circular term unification*
- *module qualification* affects *clause/2*, *retract/2* etc.
- is *caret defined*? `X^Goal :- Goal.`
- others need by the test framework (e.g. *abolish_static/1*)

Typical problems in implementations

Problems occuring multiple times in one or more implementations:

- improper *error handling:* failing, succeeding (!) or crashing instead of throwing an error
- throwing the *wrong error term* on error – various subtypes
- bad *stream handling:* not setting or returning properties, using wrong default properties, returning a stream alias (e.g. *user_input*) instead of a stream term
- *setof/3* and *bagof/3* looking for `^` in the middle of the goal
- confusion between *atom_codes* and *atom_chars*
- reporting the *wrong culprit* in goals, lists and tails
- *flag inconsistencies:* missing flags, bad flag defaults, bad flag values (e.g. *false* intead of *off*)

. . . problems in implementations (2)

- various *problems with term input* (*read/1*)
- *reliablity problems:* some test cases make the system crash, e.g. `atom_concat(A,A,AA)` if *A* is almost too long
- *clause/2* returns clause bodies in *preprocessed form:* not only module-qualified, but some code inlined
- bad *arithmetic error generation:* e.g. `nan is sqrt(-1)` does not throw an error, but it should
- bad *negative number handling* in division and remainder
- improper *rounding* or conversion of float to integer in expressions
- allowing the *zero-character* in atoms
- bad *consumption of layout characters*: *read/1* consumes space after the dot, or *number_codes* consumes heading space

Problems in the standard

We have found

- *list inconsistencies:* like omitting an error term for *is/2* from the list of allowed error terms
- *typos in examples:* like "`0 is 7//35` should succeed"
- *example inconsistencies:* like should "`call((write(3),3))`" write anything? (no)
- missing: what to do with an *uncaught error*?
- ambigious: should `write_canonical(1)` emit `'.'(1,[])` with or without quotes? The general rule is that "expressions emitted must be able to be read back unambiguously". But the meaning of the dot depends on whether it is followed by a layout character.
- missing: what to do with *non-ASCII characters* (`X = á`)?

More details in the article.

Conclusion and future work

- We have a *test framework* ported to 6 Prolog implementation. We are developing a semi-automatic tool to make porting easier for implementors. We are also documenting the framework so it can be used more easily by others.
- We have a comprehensive *test suite* covering Part 1 of the standard. We are improving it (eliminating false failures etc.). We'll extend it as soon as new parts of the standard are *implemented*.
- *Dangerous* test cases are now commented out by hand. This should be done with *implementation aspect declarations*. Discovery of dangerous test cases could be automated.
- Many implementations cannot *read terms* properly, thus they cannot read the test suite itself. Thus we should supply our own, compliant *read/1* implementation.