A short user-guide to the ismt tactic

Levent Erkök John Matthews
levent.erkok@galois.com
matthews@galois.com

Galois, Inc.
421 SW 6th Ave. Suite 300
Portland, OR 97204

(First release: May 19th, 2008
Last updated: March 24th, 2011)

Contents

1 A note on which yices to use 1

2 A note on Isabelle's oracle mechanism 2

3 Using the ismt tactic 2
3.1 Choosing the solver backend 3
3.2 Choosing the model behavior 3
3.3 Saving the output file 4
3.4 Turning on/off type-checking 5
3.5 Debugging: Tracing ismt 5
3.6 Reporting statistics 6
3.7 Argument order and defaults 6

4 Supported constructs 6

5 License 8

1 A note on which yices to use

The `yices` executable that we predominantly use the `ismt` tactic with must be in your path. Note that the latest release of `yices` (version 2.0) does *not* support Yices's internal language, so you will need a 1-series release of `yices` for `ismt` to work. (We have tested against version 1.0.29.) If you maintain multiple copies of `yices` on your computer, then you can use set the environment variable

ISMT_YICES to tell the ismt tactic precisely which yices executable to use. The following is an example setting, adjust the path accordingly on your platform:

```
$ echo ${ISMT_YICES}
/usr/local/yices-1.0.29/bin/yices
```

By using this environment variable you can have multiple versions of yices happily coexisting on your computer, letting ismt use the correct version. If this variable is not set, then ismt will simply use the name yices, which should be in your path and must be a 1.X release.

2 A note on Isabelle's oracle mechanism

Prior to Isabelle-2009 releases, Isabelle used to tag theorems that are proved via an oracle (such as ismt) with the mark [!], clearly indicating that the theorem is "trusted." This has changed in recent versions of Isabelle, where Proof-General no longer displays this mark next to the theorems proved via oracles. This can give a false sense of security, in particular if you have used ismt in the past with older versions of Isabelle where you do expect to see the trusted mark. It turns out that Isabelle still keeps track of the oracle status, it's just a matter of not displaying it directly. The reasoning behind this change is explained in the following message:

https://lists.cam.ac.uk/mailman/htdig/cl-isabelle-users/2009-April/msg00079.html

In summary, one can still query the status of a theorem to ensure whether an oracle was used in proving it (as it would be the case with ismt), although Proof-General no longer shows the mark [!] to make it immediately obvious.

3 Using the ismt tactic

The ismt tactic has been designed to have a flexible user interface, allowing the user to make various choices. It allows the following selections:

- *Solver backend:* Currently we only provide a backend translator for the Yices SMT solver, however the tactic is designed so that other solvers can be plugged-in by providing an appropriate translator.
- *Choosing the model behavior:* The tactic can be instructed to either abort the proof or fail silently when the underlying solver generates a satisfying model for the negation of the input.
- *Saving the output file:* The user can instruct the ismt tactic to save its output, mostly useful for debugging/inspection purposes.
- *Turning on/off type-checking:* Some SMT solvers can be instructed to perform type-checking on their input, or skip this step for efficiency purposes. The ismt tactic allows the user to specify the type-checking mode as an option.

- *Turning on/off debug information:* If required, the `ismt` tactic can be instructed to produce a running narrative during translation, mainly useful for debugging/inspection purposes.

- *Statistics reporting:* The `ismt` tactic can provide run-time information on the translation and backend components of the system, useful for benchmarking purposes.

In the following, we detail these parameters and show how they are used within the interactive theorem proving environment of Isabelle.

3.1 Choosing the solver backend

The back-end solver to use can be selected using the `solver` flag:

- *Flag:* `solver`

- *Possible value:* `Yices`

- *Example:*

```
by (ismt solver: Yices)
```

- *Description:* Uses the specified solver as the underlying SMT solver.

- *Default:* If not specified, `Yices` will be assumed.

- *Remarks:* The `yices` executable (downloadable on the web [1]) should be in user's path. Also see the note about `ISMT_YICES` environment variable above.

Note that this flag is currently redundant since Yices is the only backend we support for the time being.

3.2 Choosing the model behavior

This option controls how the tactic behaves if the underlying solver returns a satisfying assignment for the negation of the input, i.e., a counter-example.

- *Flag:* `model`

- *Possible values:* `silent`, `notify`, `abort`

- *Examples:*

```
by (ismt model: silent)
by (ismt model: abort)
```

- *Description:* In the `silent` mode, the proof will fail by returning the Isabelle empty sequence `Seq.empty`, with no further diagnostics. The `notify` mode is similar, except the counter-example will be printed in the `*trace*` buffer of Isabelle. If `abort` is chosen, an exception will be thrown, failing the proof attempt.

  The main use cases for the `silent` and `notify` modes are in combination with other tactics. For instance, a typical application of `ismt` could be in combination with several other rewrite rules, as in the following example:

  ```
  by (  rule VC_rules
      | simp only: VC_simps main_def Let_def
      | ismt model: silent)+
  ```

  (with appropriate definitions of `VC_simps` etc.). In this case we apply one round of rules and simplifications, followed by an attempt to prove by `ismt`, and repeating the process until the goal is simplified enough such that `ismt` can resolve it, or until none of the rules kick-in, hence causing the proof to fail.

- *Default:* If not specified, `notify` will be assumed.

3.3 Saving the output file

For inspection and debugging purposes, the user might need to see the script generated by the `ismt` tactic that is passed along to the underlying SMT solver. This option allows specifying a file name for this output.

- *Flag:* `dump`
- *Possible values: Any valid file name, or* `-`.
- *Examples:*

  ```
  by (ismt dump: "lemma.ys")
  by (ismt dump: -)
  ```

- *Description:* If a file name is given, then the script will be saved in that file. The character '`-`' is interpreted as `stdout`, i.e., the script will be printed out directly on the screen.
- *Default:* If not specified, no dump file will be generated.
- *Remarks:* The generated output will also contain the input HOL formula, the output of the SMT solver when run, along with the counter-example translated back to HOL (if any). In this sense, it will contain enough information to create a complete record of the transaction, which is an important aspect from an evaluator's point-of-view. The generated file is also directly loadable by the underlying SMT solver, hence the corresponding run can be independently repeated by the user outside of the Isabelle process.

3.4 Turning on/off type-checking

Some SMT solvers can be instructed to perform an extra step of type-checking on their input. However, such a check can incur a performance penalty, so these SMT solvers (and in particular Yices) also allow skipping this step for enhanced performance. The `tc_on` and `tc_off` flags of the `ismt` tactic allows passing this information down to the underlying solver:

- *Flags:* `tc_on`, `tc_off`
- *Examples:*

```
by (ismt tc_on)
by (ismt tc_off)
```

- *Description:* If `tc_on` is specified, the backend solver will be instructed to perform type-checking on the generated input. If `tc_off` is given, type-checking will be turned off.
- *Default:* If not specified, `tc_on` is assumed.
- *Remarks:* Unless efficiency is paramount, this flag should be left at its default value, i.e., `tc_on`. In our test cases, we have found that the additional cost of type checking is practically negligible for the Yices SMT solver.

3.5 Debugging: Tracing `ismt`

The `ismt` tactic provides a tracing mode, which is useful for debugging purposes. When turned on, it will print out various run-time data. Users should typically avoid turning tracing on, as the output tends to be quite copious especially with large input formulas.

- *Flags:* `debug_on`, `debug_off`
- *Examples:*

```
by (ismt debug_on)
by (ismt debug_off)
```

- *Description:* Turns on/off tracing data output.
- *Default:* If not specified, `debug_off` is assumed.

3.6 Reporting statistics

The `ismt` tactic can report run-times of the translator itself and the backend SMT solver, which aids in benchmarking. This behavior is controlled by the `stats_on` and `stats_off` flags:

- *Flags:* `stats_on`, `stats_off`
- *Examples:*

```
by (ismt stats_on)
by (ismt stats_off)
```

- *Description:* The `stats_on` flag turns statistics reporting on, `stats_off` turns it off.
- *Default:* If not specified, `stats_off` is assumed.

3.7 Argument order and defaults

Multiple flags can be combined on the same line, or the same flag can be specified multiple times. In the latter case, the last value given will be effective. The order of the given flags is immaterial. For instance, the following calls are all valid:

```
by (ismt model: silent debug_on dump: "a.ys" stats_on)
by (ismt dump: "a.ys" dump: "b.ys")
```

In the last example the `dump` flag is repeated. The tactic will use the second flag, hence the output will be placed in the file `"b.ys"`. It is also possible to call `ismt` with no arguments at all:

```
apply ismt
by ismt
```

In this case the defaults will be used for all the settings, as described in the preceding sections.

4 Supported constructs

Below, we describe the types, constants, and other HOL constructs that are properly understood and translated by the Yices backend. Any other construct will go uninterpreted, i.e., it will be translated as an uninterpreted constant in the Yices backend with the correct type.

- *Types.* The following types are supported:
  - Ground types: `int`, `nat`, `bool`.

  - Basic HOL types: (polymorphic) lists, option type, tuples (of arbitrary arity, including `unit`).
  - Records. (Extensible records are not supported.)
  - Types introduced via datatype declarations. These types can be parameterized and recursive. However, they cannot be mutually recursive, either directly or indirectly via nesting.
  - Functions. Note that functions *can* be higher-order.

- *Constants.* The following HOL constants are supported:[1]
  - Equality: `=`. (Polymorphic at all types.)
  - Boolean operators: `True`, `False`, $\leq$, $<$, $\longrightarrow$, $\Longrightarrow$, $\vee$, $\wedge$, $\neg$, `dvd`.
  - Arithmetic operators: $+$, $-$, $\times$, $/$, $-$ (unary minus), `div`, `mod`, `abs`, `Suc`, `min`, `max`.
  - Other: `fst`, `snd`.

- *Expressions and binding constructs.* The following constructs are supported:
  - If expressions,
  - Let bindings,
  - Lambda abstractions,
  - Quantifiers: $\forall$, $\exists$, $\bigwedge$
  - Case expressions (over tuples, naturals, option type, lists, and arbitrary user defined types),
  - Function update syntax,
  - Record update syntax.

[1] The arithmetic operators ($+$, $-$, etc.), and comparisons (`<=`, `<`) are supported both at their `int` and `nat` instances. Use of arithmetic operators at other Isabelle numeric types will remain uninterpreted. Also note that Yices only supports linear-arithmetic i.e., multiplication/division can only be done by a constant. If a non-linear expression is given to the translator, it will still be translated but Yices will reject the input with a failure message.

5 License

Copyright ©2007-2010, Galois Inc. All rights reserved.
Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- The name of Galois, Inc. may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

References

[1] Yices web site. `http://yices.csl.sri.com/`.