tuProlog with exceptions

Enrico Denti

December 16, 2010

Abstract

This document describes the new support for ISO Prolog exceptions in tuProlog 2.2. Please notice that this document is not intended to replace a tutorial nor a full manual about exception support in Prolog: rather, it means to provide a short yet effective reference to the exception support added in this tuProlog version.

1 Exceptions in ISO Prolog

The ISO Prolog standard (ISO/IEC 13211-1) has been published in 1995. Among the many additions, it introduces the `catch/3` e `throw/1` constructs for exception handling. The first distinction has to be made between *errors* and *exceptions*. An error is a particular circumstance that interrupts the execution of a Prolog program: when a Prolog engine encounters an error, it raises an exception. The exception handling support is supposed to intercept the exception and transfer the execution flow to a suitable exception handler, with any relevant information. Two basic principles are followed during this operation:

- *error bounding* – an error must be bounded and not propagate through the entire program: in particular, an error occurring inside a given component must either be captured at the component's frontier, or remain invisible and be reported nicely. According to ISO Prolog, this can be done via the `catch/3` predicate.
- *atomic jump* – the exception handling mechanism must be able to exit atomically from any number of nested execution contexts. According to ISO Prolog, this is done via the `throw/1` predicate.

In practice, `throw(`*`Error`*`)` raises an exception, while the controlled execution of a goal is launched via the `catch(`*`Goal, Catcher, Handler`*`)` predicate, which is very much like the `try/catch` construct of many imperative languages. Here, *`Goal`* is first executed: if an error occurs, the subgoal where the error occurred is replaced by the corresponding `throw(`*`Error`*`)`, which raises the exception. Then, a matching `catch/3` clause – that is, a clause whose second argument unifies with *`Error`* – is searched among the antenate nodes in the resolution tree: if one is found, the path in the resolution tree is cut, the catcher itself is removed (because it only applies to the protected goal, not to the handler), and the *`Handler`* predicate is executed. If, instead, no such matching clause is found, the execution simply fails. So, `catch(`*`Goal, Catcher, Handler`*`)` performs exactly like *`Goal`* if no exception are raised: otherwise, all the choicepoints generated

by *Goal* are cut, a matching *Catcher* is looked for, and if a one is found then *Handler* is executed, maintaining the substitutions made during the previous unification process. In the very end, the execution continues with the subgoal which follows `catch/3`. However, any side effects possibly occurred during the execution of a goal are not undone in case of exceptions, exactly as it normally happens when a predicate fails. Summing up, `catch/3` is true if:

- `call(`*Goal*`)` is true;

  or

- `call(`*Goal*`)` is interrupted by a call to `throw(`*Error*`)` whose *Error* unifies with *Catcher*, and the subsequent `call(`*Handler*`)` is true;

If *Goal* is non-deterministic, it can obviously be executed again in backtracking. However, it should be clear that *Handler* *is possibly executed just once*, since all the choicepoints of *Goal* are cut in case of exception.

1.1 Examples

As a first, basic example, let us consider the following toy program:

```
p(X):- throw(error), write('---').
p(X):- write('+++').
```

and let us consider the behaviour of the program in response to the execution of the goal:

```
?:- catch(p(0), E, write(E)), fail.
```

which tries to execute `p(0)`, catching any exception `E` and handling the error by just printing it on the standard output (`write(E)`).

Perhaps surprisingly, the program will just print `'error'`, not `'error---'` or `'error+++'`. The reason is that once the exception is raised, the execution of `p(X)` is aborted, and after the handler terminates the execution proceeds with the subgoal which follows `catch/3`, i.e. `fail`. So, `write('---')` is never reached, nor is `write('+++')` since all the choicepoints are cut upon exception.

In the following we report a small yet complete set of mini-examples, thought to put in evidence one single aspect of tuProlog compliance to the ISO standard.

Example 1: *`Handler` must be executed maintaining the substitutions made during the unification process between `Error` and `Catcher`*

Program: `p(0) :- throw(error).`
Query: `?- catch(p(0), E, atom_length(E, Length)).`
Answer: `yes.`
Substitutions: `E/error`, `Length/5`

Example 2: *the selected `Catcher` must be the nearest in the resolution tree whose second argument unifies with `Error`*

Program:
```
p(0) :- throw(error).
p(1).
```
Query: `?- catch(p(1), E, fail), catch(p(0), E, true).`
Answer: `yes.`

Substitutions: `E/error`

Example 3: *execution must fail if an error occurs during a goal execution and there is no matching* `catch/3` *predicate whose second argument unifies with* *`Error`*

Program: `p(0) :- throw(error).`
Query: `?- catch(p(0), error(X), true).`
Answer: `no.`

Example 4: *execution must fail if* *`Handler`* *is false*

Program: `p(0) :- throw(error).`
Query: `?- catch(p(0), E, false).`
Answer: `no.`

Example 5: *if* *`Goal`* *is non-deterministic, it is executed again on backtracking, but in case of exception all the choicepoints must be cut, and* *`Handler`* *must be executed only once*

Program:
```
p(0).
p(1) :- throw(error).
p(2).
```
Query: `?- catch(p(X), E, true).`
Answer: `yes.`
Substitutions: `X/0`, `E/error`
Choice: `Next solution?`
Answer: `yes.`
Substitutions: `X/1`, `E/error`
Choice: `Next solution?`
Answer: `no.`

Example 6: *execution must fail if an exception occurs in* *`Handler`*

Program: `p(0) :- throw(error).`
Query: `?- catch(p(0), E, throw(err)).`
Answer: `no.`

1.2 Error classification

So far we have just said that, when an exception is raised, `throw(`*`Error`*`)` is executed, and a matching `catch/3` is looked for, but no specifications have been given about the possible structure of the *`Error`* term. According to the ISO Prolog standard, such a term should follow the pattern `error(`*`Error_term`*`,` *`Implementation_defined_term`*`)` where *`Error_term`* is constrained by the standard to a pre-defined set of possible values, in order to represent the error category: *`Implementation_defined_term`*, instead, is left for implementation-specific details, and could also be omitted.

The error classification induced by *`Error_term`* is flat, so as to easily support pattern matching. Ten error classes are identified by the ISO standard:

1. `instantiation_error`: when the argument of a predicate or one of its components is a variable, while it should be instantiated. A typical example is `X is Y+1` if `Y` is not instantiated when `is/2` is evaluated.

2. `type_error(ValidType, Culprit)`: when the type of an argument of a predicate, or one of its components, is instantiated, but nevertheless incorrect. In this case, *ValidType* represents the expected data type (one of `atom`, `atomic`, `byte`, `callable`, `character`, `evaluable`, `in_byte`, `in_character`, `integer`, `list`, `number`, `predicate_indicator`, `variable`), while *Culprit* is the wrong type found. For instance, if a predicate operates on dates and expects months to be represented as integers between 1-12, calling the predicate with an argument like `march` instead of `3` would raise a `type_error(integer, march)`, since an integer was expected and `march` was found instead.

3. `domain_error(ValidDomain, Culprit)`: when the argument type is correct, but its value falls outside the expected range. *ValidDomain* is one of `character_code_list`, `close_option`, `flag_value`, `not_empty_list`, `not_less_than_zero`, `io_mode`, `operator_priority`, `operator_specifier`, `prolog_flag`, `read_option`, `source_sink`, `stream`, `stream_option`, `stream_or_alias`, `stream_position`, `stream_property`, `write_option`. In the example above, a domain error could be raised if, for instance, a value like 13 was provided for the month argument.

4. `existence_error(ObjectType, ObjectName`: when the referenced object to be accessed does not exist. Again, *ObjectType* is the type of the unexisting object, and *ObjectName* its name. *ObjectType* is `procedure`, `source_sink`, or `stream`. If, for instance, the file `'usr/goofy'` does not exist, an `existence_error(stream, 'usr/goofy')` would be raised.

5. `permission_error(Operation, ObjectType, Object)`: when *Operation* is not allowed on *Object*, which is of type *ObjectType*. *Operation* is one of `access`, `create`, `input`, `modify`, `open`, `output`, or `reposition`, while *ObjectType* falls among `binary_stream`, `operator`, `past_end_of_stream`, `private_procedure`, `static_procedure`, `source_sink`, `stream`, `flag`, and `text_stream`.

6. `representation_error(Flag)`: when an implementation-defined limit, whose category is given by *Flag*, is violated during execution. *Flag* is one of `character`, `character_code`, `in_character_code`, `max_arity`, `max_integer`, `min_integer`.

7. `evaluation_error(Error)`: when the evaluation of a function produces an exceptional value. Accordingly, *Error* is one of `float_overflow`, `int_overflow`, `undefined`, `underflow`, `zero_divisor`.

8. `resource_error(Resource)`: when the Prolog engine does not have enough resources to complete the execution of the current goal. Typical examples are the reach of the maximum number of opened files, no further available memory, etc. Accordingly, `resource_error(Resource)` can be any valid term.

9. `syntax_error(Message)`: when external data, read from an external source, have an incorrect format or cannot be processed for some reason. This kind of error typically occurs during *read* operations. *Message* can be any valid (simple or compound) term describing the occurred problem.

10. `system_error`: this latter category represents any other unexpected error which does not fall in any of the above categories.

2 Implementing Exceptions in tuProlog

Implementing exceptions in tuProlog does not mean just to extend the engine to support the above mechanisms: given its library-based design, and its intrinsic support to multi-paradigm programming, adding exceptions in tuProlog has also meant (1) to revise all the existing libraries, modifying any library predicate so that it raises the appropriate type of exception instead of just failing; and (2) to carefully define and implement a model to make Prolog exceptions not only co-exist, but also fruitfully operate with the Java (or C#/.NET) imperative world, which brings its own concept of exception and its own handling mechanism.

As a preliminary step, the finite-state machine which constitutes the core of the tuProlog engine was extended with a new *Exception* state, between the existing *Goal Evaluation* and *Goal Selection* states. Then, all the tuProlog libraries were revised, according to clearness and efficiency criteria — that is, the introduction of the new checks required for proper exception raising should not reduce performance unacceptably. This issue was particularly relevant for runtime checks, such as `existence_error`s or `evaluation_error`s; moreover, since tuProlog libraries could also be implemented partly in Prolog and partly in Java, careful choices had to be made so as to introduce such checks at the most adequate level in order to intercept all errors while maintaining code readability and overall organisation, while guaranteeing efficiency. This led to intervene with extra Java checks for libraries fully implemented in Java, and with new "Java guards" for predicates implemented in Prolog, keeping the use of Prolog meta-predicates (such as `integer/1`) to a minimum.

With respect to the third aspect, which will be discussed more in depth below, one key aspect to be put in evidence right now concerns the handling of Java objects accessed from the Prolog world via Javalibrary. At a first sight, one might think of re-mapping Java exceptions and constructs onto the Prolog one, but this approach is unsatisfactory for three main reasons:

- the semantics of the Java mechanism should not be mixed with the Prolog one, and vice-versa;
- the Java construct admits also a `finally` clause which has no counterpart in ISO Prolog;
- the Java catching mechanisms operates hierarchically, while the `catch/3` predicate operates via pattern matching and unification, allowing for multiple granularities.

For these reasons, supporting Java exceptions from tuProlog programs called for two further, *ad hoc* predicates *which are not present in ISO Prolog* because ISO Prolog does not consider multi-paradigm programming: `java_throw/1` and `java_catch/3`.

2.1 Java exceptions from tuProlog

The `java_throw/1` predicate has the form

`java_throw(`*JavaException*`(`*Cause*`, `*Message*`, `*StackTrace*`))`

where *JavaException* is named after the specific Java exception to be launched (e.g., `'java.io.FileNotFoundException'`, and its three arguments represent the typical properties of any Java exception. More precisely, *Cause* is a string representing the cause of the exception, or `0` if the cause is unknown; *Message* is the message associated to the error (or, again, `0` if the message is missing); *StackTrace* is a list of strings, each representing a stack frame.

The `java_catch/3` predicate takes the form

`java_catch(`*JGoal*`, [(`*Catcher1*`, `*Handler1*`),`
`...,`
`(`*CatcherN*`, `*HandlerN*`)], `*Finally*`)`

where *JGoal* is the goal (representing a Java operation in the Java world) to be executed under the protection of the handlers specified in the subsequent list, each associated to a given type of Java exception and expressed in the form `java_exception(`*Cause*`, `*Message*`, `*StackTrace*`)`, with the same argument semantics explained above. The third argument *Finally* expresses the homonomous Java clause, and therefore represents the predicate to be executed at the very end either of the *Goal* or one of the *Handler*s. If no such a clause is actually needed, the conventional atom (`'0'`) has to be used as a placeholder.

The predicate behaviour can be informally expressed as follows. First, *JGoal* is executed. Then, if no exception is raised via `java_throw/1`, the *Finally* goal is executed. If, instead, an exception is raised, all the choicepoints generated by *JGoal* (in the case of a non-deterministic predicate like `java_object_bt/3`, of course) are cut: if a matching handler exists, such a handler is executed, maintaining the variable substitutions. If, instead, no such a handler is found, the resolution tree is backsearched, looking for a matching `java_catch/3` clause: if none exists, the predicate fails. Upon completion, the *Finally* part is executed anyway, then the program flow continues with the subgoal following `java_catch/3`. As already said above, side effects possibly generated during the execution of *JGoal* are *not* undone in case of exception.

So, summing up, `java_catch/3` is true if:

- *JGoal* and *Finally* are both true;

  or

- `call(`*JGoal*`)` is interrupted by a call to `java_throw/1` whose argument unifies with one of the *Catcher*s, and both the execution of the catcher and of the *Finally* clause are true.

Even if *JGoal* is a non-deterministic predicate, like `java_object_bt/3`, and therefore the goal itself can be re-executed in backtracking, in case of exception only one handler is executed, then all the choicepoints generated by *JGoal* are removed: so, no further handler would ever be executed for that exception. In other words, `java_catch/3` only protects the execution of *JGoal*, *not* the handler execution or the *Finally* execution.

2.2 Examples

First, let us consider the following program:

```
?- java_catch(java_object('Counter', ['MyCounter'], c),
      [('java.lang.ClassNotFoundException'(Cause, Msg, StackTrace),
        write(Msg))],
      write(+++)).
```

which tries to allocate an instance of `Counter`, bind it to the atom `c`, and – if everything goes well – print the `'+++'` message on the standard output. Indeed, this is precisely what happens if, at runtime, the class `Counter` is actually available in the file system. However, it might also happen that, for some reason, the required class is *not* present in the file system when the above predicate is executed. Then, a `'java.lang.ClassNotFoundException'(Cause, Msg, StackTrace)` exception is raised, no side effects occur – so, no object is actually created – and the *`Msg`* is printed on the standard output, followed by `'+++'` as required by the *`Finally`* clause. Since the *`Msg`* in this exception is the name of the missing class, the global message printed on the console is `Counter+++`.

In the following we report a small yet complete set of mini-examples, thought to put in evidence one single aspect of tuProlog compliance to the ISO standard.

Example 1: *the handler must be executed maintaining the substitutions made during the unification process between the exception and the catcher: then, the* ***`Finally`*** *part must be executed.*

```
?- java_catch(java_object('Counter', ['MyCounter'], c),
      [('java.lang.ClassNotFoundException'(Cause, Message, _),
        X is 2+3)], Y is 2+5).
```

Answer: `yes.`
Substitutions: `Cause/0`, `Message/'Counter'`, `X/5`, `Y/7`.

Example 2: *the selected* ***`java_catch/3`*** *must be the nearest in the resolution tree whose second argument unifies with the launched exception*

```
?- java_catch(java_object('Counter', ['MyCounter'], c),
      [('java.lang.ClassNotFoundException'(Cause, Message, _),
        true], true),
   java_catch(java_object('Counter', ['MyCounter2'], c2),
      [('java.lang.ClassNotFoundException'(Cause2, Message2, _),
        X is 2+3], true).
```

Answer: `yes.`
Substitutions: `Cause/0`, `Message/'Counter'`, `X/5`, `C/0`, `Message2/'Counter'`.

Example 3: *execution must fail if an exception is raised during the execution of a goal and no matching* ***`java_catch/3`*** *can be found*

```
?- java_catch(java_object('Counter', ['MyCounter'], c),
     [('java.lang.Exception'(Cause, Message, _), true)], true)).
```

Answer: `no.`

Example 4: ***`java_catch/3`*** *must fail if the handler is false*

```
?- java_catch(java_object('Counter', ['MyCounter'], c),
     [('java.lang.Exception'(Cause, Message, _), false)], true)).
```

Answer: `no.`

Example 5: *`java_catch/3` must fail also if an exception is raised during the execution of the handler*

```
?- java_catch(java_object('Counter', ['MyCounter'], c),
     [('java.lang.ClassNotFoundException'(Cause, Message, _),
       java_object('Counter', ['MyCounter'], c))], true).
```

Answer: `no.`

Example 6: *the `Finally` must be executed also in case of success of the goal*

```
?- java_catch(java_object('java.util.ArrayList', [], 1),
      [E, true], X is 2+3).
```

Answer: `yes.`
Substitutions: `X/5`.

Example 7: *the `Handler` to be executed must be the proper one among those available in the handlers' list*

```
?- java_catch(java_object('Counter', ['MyCounter'], c),
    [('java.lang.Exception'(Cause, Message, _), X is 2+3),
     ('java.lang.ClassNotFoundException'(Cause, Message, _), Y is 3+5)],
     true).
```

Answer: `yes.`
Substitutions: `Cause/0`, `Message/'Counter'`, `Y/8`.

3 Library predicates

All tuProlog library predicates have been revised so as to raise the proper exceptions, instead of failing, as specified by ISO standards. Please refer to such standards for specific information on this topic.

With respect to tuProlog multi-paradigm programming, supported by JavaLibrary, two further predicates have been introduced *beyond* the ISO standard discussed above: `java_throw/1` and `java_catch/3`. These predicates work similarly to the standard `throw/1` and `catch/3` predicates, but refer to exceptions raised inside the Java world, instead of Prolog-raised ones. More info on this topic will be added in a future version of this document. However, some (hopefully clear) usage examples can be found in the test source file `JavaLibraryExceptionsTestCase.java` in the `exceptions-test` folder.