PX1

February 10, 2022

1 PX: Using XSB in Python

1.1 1: Getting Started

This notebook provides some examples of how to run XSB from a Python interpreter, and by extension, how to embed XSB into Python applications using an XSB package tentatively called **px** The material presented here does not cover all **px** functionality, and does not even begin to cover all XSB functionality. See the XSB manuals for background in and full details of XSB, and Volume 2 Chapter 18 for details of **px**.

To start, simply import the **px** module like any other Python module.

[ ]:
```
from px import *
```

Note that importing **pyxsb** automatically starts up XSB within the same process as your Python session. You can get basic information on commands via help(), but see the XSB manual for full details (Volume 2, Chapter 18, remember)

[ ]:
```
help('px')
```

1.2 2. Deterministic Queries and Commands

Let's get started with a simple query to XSB to reverse a list containing a integers, a tuple and a dictionary. `basics` is the XSB module, and `reverse` is the XSB predicate.

[ ]:
```
Ans,TV = px_qdet('basics','reverse',[1,2,3,('t1','t2'),{'a':{'b':'c'}}])
print(Ans)
print(TV)
```

We'll discuss below why `px_qdet()` returns both an *answer* and a *truth value*. First we note that alternatively, we could have called `pp_px_qdet()`which pretty-prints both the query and answer. For instance:

[ ]:
```
pp_px_qdet('basics','reverse',[1,2,3,('t1','t2'),{'a':{'b':'c'}}])
```

Various **px** functions have a pretty printing form, and we'll use the `pp_px` and `px` prefixes interchangeably.

As seen above, `px_qdet()` adds an extra argument to the Prolog call so that `reverse/2` is called. The extra argument allows bindings to be passed back to Prolog. But suppose you want to call the Prolog goal `consult(px)` or `p(a,b,c)`. In this case, just use `px_cmd()` -- or `pp_px_cmd()`:

[ ]:
```
pp_px_cmd('px_test','one_ary_fail','p')
```

In this case, there is no answer to return, but the truth value indicates that the query failed (in Prolog, a failure is different than an error condition, as we'll see below).

In general, truth values passed back by XSB can be:

- 1 (*true*) which means the query succeeded and that the answer is true in the Well-Founded Model of the program.
- 0 *false* which means that the query failed and that the query has no answers in the Well-Founded Model of the program.
- 2 *undefined* which means that the query succeeded, but the answer is neither *true* nor *false* in the Well-Funded Mdel of the program.

An example of returning an answer with *undefined* truth value is:

[ ]:
```
pp_px_cmd('px_test','win','p')
```

Using px_cmd() to consult XSB files was mentioned above; this can be done fully interactively with **px**. Let's say you made a change to **px_test.P**. There's no need to leave your session -- just (re-)consult it.

[ ]:
```
consult('px_test')
```

One least aspect of querying is exception handling. If an exception occurs in XSB execution, it is caught by **px**, a Python exception of the general **Exception** class is then raised for Python to handle. However, by another call to XSB the actual XSB exception can also be examined. Within the pretty-print display format this looks like:

[ ]:
```
pp_px_qdet('usermod','open','missing_file','read')
```

3. List and Set Comphrehensions with PX

The above queries were determinstic, but you can collect all solutions of a non-deterministic query using a construct similar to list or set comprehension in Python.By default the collection is a Python list, but it can also be a Python set.

The Python function call

px_comp(Module,Predicate,*args,kwargs)**

returns a collection of answers such that the Prolog goal

Module:Predicate(input_args,output_variables)

The number of variable arguments in the Python call (i.e., the number of elements in the *args tuple) corresponds to the number of **input_arguments** in the XSB goal. The number of **output_variables** is by default 1, but can be set by the **vars** keyword argument.

Let's see how this works. Consider the simple predicate:

```
test_comp(a).
test_comp(b).
test_comp(c).
test_comp(d).                                                    test_comp(e):- unk(something
```

[ ]:
```
pp_px_comp('px_test','test_comp')
```

Note that when translating from Prolog to Python a Prolog term in `px_comp()` answer bindings **test_comp/1** are returned. Suppose the predicate `test_comp` were in fact 2-ary:

test_comp(a,1). test_comp(b,2). test_comp(c,3). test_comp(d,4). test_comp(e,5):- unk(something). test_comp(e,5):- unk(something_else).

In this case we just set the `vars` keyword:

[ ]:
```
pp_px_comp('px_test','test_comp',vars=2)
```

Now the bindings for the 2 variables are captured by a 2-ary tuple. To see why the first two answers appear the same, execute the follwoing:

[ ]:
```
pp_px_comp('px_test','test_comp',vars=2,truth_vals=DELAY_LISTS)
```

Just as a reminder, the answers are being passed back to Python which are then output as terms via the **px** Pyrhon pretty-printer. (The Python representation of Prolog structures is discussed in the manual.)

If you just want to remove duplicates, simply use set comprehension instead.

[ ]:
```
pp_px_comp('px_test','test_comp',vars=2,set_collect=True)
```

The default behavior of `px_comp()`is to return a list, list comprehension is more general. Python sets cannot contain dictionaries or other sets.

2 Going Forward

As mentioned, a short notebook is only a sampler of what XSB can do. The file **tpx()** in the directory **XSB_ROOT/XSB/packages/xsbpy/px** contains a number of other examples in its test file **tpx.py**. These examples include constraint-based reasoning, timed calls, and stress tests. Probabilistic reasoning and virtually all other XSB functionality is also supported -- well, pretty much. See Volume 2 chapter 18 for a list of current limitiations, most all of which are in the process of being addressed.