OCaml Under The Hood: SmartPy

Seb Mondet[*], TQ Tezos[†]

Abstract

SmartPy is a complete system to develop smart-contracts for the Tezos blockchain. It is an embedded EDSL in python to write contracts and their tests scenarios. It includes an online IDE, a chain explorer, and a command line interface. Python is used to generate programs in an inperative, type infered, intermediate language called SmartML. SmartML is also the name of the OCaml library which provides an interpreter, a compiler to Michelson (the smart-contract language of Tezos), as well as a scenario "on-chain" interpreter. The IDE uses a mix of OCaml built with `js_of_ocaml` and pure Javascript. The command line interface also builds with `js_of_ocaml` to run on Node.js.

Introduction

Tezos is a blockchain project well known of the OCaml ecosystem. It has support for rich smart-contracts, i.e. accounts of "programmable money," which are the basis for distributed applications ("DApps") which use the blockchain as a synchronization source between non-trusting parties.

Michelson, the smart-contract language specified in the current Tezos protocol, is a quite low-level, stack-based, statically typed, language. Even though some aficionados do love programming in Michelson, most users prefer to avoid it and, hence, quite a few higher-level language projects have appeared in the ecosystem, e.g., SmartPy[1], LIGO[2], Lorentz[3], Juvix[4], or Archetype[5].

SmartPy is one of these projects but should be seen more as a meta-programming framework than a (new) high-level language. It uses some of python's annotation features as well as some light (optional) syntactic sugar to provide a "native feel" but it is essentially a library of constructors for SmartML programs and test scenarios. Here is a minimal example:

```
import smartpy as sp
class HelloWorld(sp.Contract):
    def __init__(self): self.init(mem = "")

    @sp.entryPoint
    def remember(self, param):
        self.data.mem += param

@addTest(name = "Test")
def test():
    scenario = sp.test_scenario()
    scenario.h1("Let's Go!")
    c = HelloWorld()
    scenario += c
    scenario += c.remember("Hello World")
```

The SmartPy team publishes the software under the MIT-license periodically at https://gitlab.com/SmartPy/smartpy, various builds ("stable"/"dev"/"test") of the distribution are also provided to users.

Why Python?

Python is one of the most popular languages in the world:

- It has an intuitive syntax for most people.
- It provides good meta-programming capabilities (e.g. programmable AST annotations).
- New users (believe they) already know how to program with it, they can focus on learning the really hard part: smart-contract design and development.

SmartPy also provides a very complete online IDE as well as a command-line application. Both allow the user to compile, simulate, run (sandboxes and test-networks) and inspect the contracts.

Implementation

SmartPy programs use Python essentially as a very powerful macro-language for *SmartML*. SmartML is an inperative, fully type inferred, intermediate language which is defined in an OCaml library. The library itself contains a

[*] http://seb.mondet.org
[†] https://tqtezos.com
[1] https://smartpy.io
[2] https://ligolang.org/
[3] https://gitlab.com/morley-framework/morley
[4] https://github.com/cryptiumlabs/juvix/
[5] https://archetype-lang.org/

type-inference engine, an interpreter, an optimizing compiler to Michelson, as well as a scenario "on-chain" runner.

The WebIDE uses a mix of OCaml built with `js_of_ocaml` and pure Javascript. The command line interface is also buildt with `js_of_ocaml` in order to run with Node.js and make distribution much easier.

The user edits their code written from scratch or, most likely, imported from one of the many self-documenting "templates" provided by the UI. The workflow of what happens when a user clicks on "Run & Test" in WebIDE is as follows:

- The Python code is interpreted with Brython[6].
- It constructs a SmartML expressions and scenarios.
- The contract and the tests enter the `js_of_ocaml` world:
  - Full type inference and type checking.
  - Simulation of the tests in the interpreter.
  - Compilation to Michelson.
  - Back to the UI to display the results.

The interpreter requires fast execution of the contracts in a command line application and in the WebIDE. Moreover, for interoperability some primitive operations of Michelson must match bit-for-bit what the interpreter computes on both execution environments. This includes all the cryptographic operations and the binary serialization of values.

For instance, a user should be able to use other tools to construct the Ed2551 signatures fed to the entry-point in the example below. The behavior of the call to `check_signature` as well as the serialization (`sp.pack`) should match perfectly the specification and implementation of the Tezos protocol:

```
@sp.entryPoint
def set_current_value(self, params):
    thing_to_sign = sp.pack(
        sp.record(
            o = self.data.current_value,
            n = params.new_value,
            a = sp.self,
            c = self.data.counter))
    sp.verify(
        sp.check_signature(
            self.data.boss_public_key,
            params.user_signature,
            thing_to_sign))
    self.data.current_value = params.new_value
    self.data.counter = self.data.counter + 1
```

As current work-in-progress or future work items, our roadmap contains:

- Decompilation from Michelson to SmartPy.
- Other static analyses, such as abstract interpretation: ownership, value domains, etc. and gas usage prediction.
- Other compilation targets, like contract storage parsing code, or proof-friendly representations (e.g. WhyML).
- An OCaml version of the EDSL, i.e. generate SmartML from OCaml instead of Python.

[6]https://brython.info/

Ecosystem and Real-World Users

SmartPy benefits from quite some popularity within the Tezos ecosystem; especially given how "niche" the product is. The main Telegram help-channel[7] has more than 200 members, and the twitter[8] account has about 600 followers.

There are already 3rd party online courses, like blockmatics.io[9] or "Cryptobots vs Aliens"[10], and most hackathons include SmartPy (e.g. CoinList[11]). Some nascent financial applications such as ChainLink[12] already build on the platform. Generic development platforms like ConseilJS[13] natively support SmartPy.

We have also participated in the development of standard contract interfaces, through reference implementations. For instance, FA2-SmartPy[14] implementation of the TZIP-12[15] standard abuses meta-programming features to provide many different "builds" of the contract corresponding to various options; it also happens to use OCaml-code generation to provide type-safe access to those smart-contract variants, the specialized command-line application, used for demoing, benchmarks, and generating documentation can safely evolve as the draft specification changes.

The Talk

The talk will present all of the above with a focus on:

- The specific OCaml aspects: the portability between browser Vs Node.js packaging, the re-implementation of lower-level Tezos functionality (including the cryptographic functions from C compiled to Javascript using Emscripten and bound with `gen_js_api`[16]).
- The meta-programming approach both in Python *and* OCaml.

[7]https://t.me/SmartPy_io/
[8]https://twitter.com/smartpy_io
[9]https://training.blockmatics.io/p/tezos-smartpy-developer-course/
[10]https://cryptocodeschool.in/tezos/
[11]https://coinlist.co/build/tezos
[12]https://chain.link/
[13]https://cryptonomic.github.io/ConseilJS/#/
[14]https://gitlab.com/smondet/fa2-smartpy
[15]https://gitlab.com/tzip/tzip
[16]https://github.com/LexiFi/gen_js_api