Integrating Verified MW

Sam Merten

1 Setup and Use

1.1 Prerequisites

See README.MD for how to install prerequisites.

1.2 Quick Start

Using the provided files, one can create and execute a simple instance of the system out of the box. From the top level:

- `$ make` – This extracts a verified instance of multiplicative weights into the OCaml files at `/runtime/extractedMW.ml` and `/runtime/extractedMW.mli`
- `$ cd runtime` – Navigate to the runtime folder in order to build and execute the system.
- `$ make` – This will compile the extracted code into the executable `envProp`, as well as compile the OCaml shim, `env_client_interface`, responsible for handling communication between the verified client and unverified environment.
- `$ ./init.sh` – This builds the named pipes over which communication runs between the processes.
- As separate processes execute: `$ ./clientProc`, `$ python3 ./envProc.py`, `$ ./env_client_interface`

1.3 Adapting the system

- **Program Parameters:** Due to quirks of Coq's module system, a desire to minimize the occurrences of non-verified code in the extracted client program, and an attempt to constrain the execution of the system to those in which the verified performance bounds are applicable, many program parameters are stored independently by the various processes.

  In particular, the learning parameter, η, as well as the number of rounds and strategies tracked by the client are stored in various forms across `clientOracle.v`, `/runtime/OTP.ml` and `/runtime/envProc.py`.

However, there is a small script at the top level to make it easy to adjust these parameters across the various files:`config.py`. By changing the variables in this file and calling `$ python3 config.py`, values for these parameters will be propagated across the various files.

- **Cost Vectors:** The file`./envProc.py` shows an example of how one can build a simple instantiation of the environment process. To use one's own cost vectors, one need only return a string representation of a list of dyadics (see the definition of dyadics below, as well as the description of envProc in Section 3.

2 Definitions

2.1 OCaml Data Types

- big_int : Arbitrary precision integers from zarith library.
- dyadic : Representation of dyadic rational numbers as pairs of big_int, where $d = (x, y)$ corresponds to $\frac{x}{2^y}$.

3 Architecture

This section outlines the various components of the integration system, as well as their interactions.

3.1 Runtime Components

During runtime, the system relies on the simultaneous execution of three distinct processes:

1. clientProc – This is an application running an extracted version of the verified MW client code.
   - **sends** to clientEnvInterface:
     (big_int, dyadic) list, pairing strategies with weights
   - **receives** from clientEnvInterface:
     (big_int, dyadic) list, pairing strategies with costs in the range [-1, 1]
2. envProc – This process acts as the environmental component of MW. During each round, it receives a weight vector from the client and is responsible for returning an associated cost vector back to the client.
   - **sends** to clientEnvInterface:
     space-separated string representation of a dyadic list, where element i corresponds to the cost incurred by strategy i in the current round.

- **receives** from CLIENTENVINTERFACE:

  space-separated string representation of a dyadic list, where element i corresponds to the weight the agent associates with strategy i in the current round.

3. CLIENTENVINTERFACE – Unverified OCaml program responsible for managing communication between clientProc and envProc.

   - **sends** to CLIENTPROC:

     A (big_int, dyadic) list, pairing strategies with costs in the range [-1, 1], generated by parsing the string received from CLIENTENVINTERFACE.
   - **receives** from CLIENTPROC:

     A (big_int, dyadic) list, representing the weight vector generated by the client in the current round.
   - **sends** to ENVPROC:

     A space-separated string representation of a float list, generated from the list received from CLIENTPROC.
   - **receives** from ENVPROC:

     A space-separated string representation of a list of dyadics representing the cost vector generated by the environment in the current round.

This architecture isolates the verified components of the system (everything performed by CLIENTPROC) from the unverified components of the system and allows for modularity in ENVPROC's actual implementation. Any program capable of communicating strings across named pipes could easily be adapted to act as an environment process for the system.

4 Communication

Communication occurs across two pairs of named pipes, one pair responsible for messages between CLIENTPROC and CLIENTENVINTERFACE, and the other pair handling messages between ENVPROC and CLIENTENVINTERFACE. Communication to/from CLIENTPROC is handled through Ocaml's marshaling functions. Communication to/from ENVPROC simply sends strings.

5 Notes On Verification

5.1 Verification guarantee

The verification result established in the Coq development states, at a high level, that provided nothing abberant occurs in during its execution, the MW program described therein is no-regret.

This is formalized by the following theorem (located in `oracleExtract.v`):

```
Lemma mwu_proof :
  ∀finState,
    MWUextract.interp
      (mult_weights
         (MWUProof.t _ _ _) num_rounds)
                 (MWUextract.init_cstate eta) = Some finState →
    ∃s' : state (MWUProof.t _ _ _) oracleState oracle_chanty,
      MWUProof.match_states _ _ _ match_states_myOracles s' finState ∧
      ((state_expCost1 (all_costs0 s') s' − OPTR a0 s') / Tx num_rounds ≤
       rat_to_R (Q_to_rat (D_to_Q eta)) +
       ln (rat_to_R #|MWUProof.t _ _ _|%:R) /
        (rat_to_R (Q_to_rat (D_to_Q eta)) ∗ Tx num_rounds))%R.
Proof.
Qed.
```

There are two major reasons for the complexity of this expression. The first stems from the fact that the communication received from the environment may be ill-formed or terminate prematurely. In these cases, the client program terminates and returns None. Since the no-regret result holds only for those instances in which the program successfully terminates, the no-regret bound is conditioned by the successful execution of the program in the lemma above. The second reason, simply has to do with explicit type conversions between the various numeric data types. The final cause for complexity has to do with the calculation of expected cost and the representation of the weights vector maintained by the client during execution.

5.2 Axiomatization and extraction

An additional issue arises when building this system arising from Coq's status as a purely functional language. Purely functional languages lack side-effects, meaning they often lack functionality such as I/O and mutable data-structures.

In order to introduce networking capability into our development we axiomatize the existence of a number of different types and functions in our Coq development. Their axiomatization are shown below:

```
  (∗∗ Input/output channel types ∗)
Axiom inChan : Type.
Axiom outChan : Type.
  (∗∗ Functions for generating a channel from a string ∗)
Axiom open_in : string → inChan.
Axiom open_out : string → outChan.
  (∗∗ Functions for sending and receiving info over channels ∗)
Axiom inChan_recv : inChan → list (M.t ∗ dyadic.D).
Axiom outChan_send : outChan → list (M.t ∗ dyadic.D) → unit.
```

Outside of the existence of such functions, along with their typing, nothing is assumed regarding their behavior. During extraction, these axiomatized functions are instantiated with actual OCaml types and definitions using Coq's **Extract Constant** directive.