README.md

CAMP: Cost-Aware Multiparty Session Protocols

This is the artifact for the paper *CAMP: Cost-Aware Multiparty Session Protocols*. The artifact comprises:

- A library for specifying cost-aware multiparty protocols.
- The raw data used for comparing the cost models with real execution costs.
- The cost-aware protocol specifications of the benchmarks that we studied.

The library for specifying cost-aware protocols also provides functions for extracting cost equations from them, and for estimating recursive protocol latencies (i.e. average cost per protocol iteration). We provide a script for extracting cost equations, and instantiating them using the parameters used in the paper.

Part 1: Getting Started

Setup Instructions

We provide a Docker image (filename `100.tgz`) with the contents of the artifact. We used Docker version 19.03.12, build 48a66213fe. Docker can be installed from the package managers of the most common Linux distributions. We refer to the docker website for further details on how to get started with Docker.

The following command will print the available images:

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
```

Assuming that the artifact image, `100.tgz`, is available in the current directory, the following command will load the image for the artifact:

```
$ docker load < 100.tgz
```

This will make the artifact image available next time we run `docker images`.

```
$ docker images
REPOSITORY              TAG                 IMAGE ID            CREATED             SIZE
oopsla20-artifact-100   latest              9a817f3071c6        10 hours ago        4.01GB
```

To run the artifact, use the command:

```
$ docker run -ti oopsla20-artifact-100
```

This will prompt a welcome message, and run a shell in the working directory where the artifact contents are located.

Remark. Alternatively, it is possible to download the artifact from the github url: https://github.com/camp-cost/camp. We used GHC compiler 8.8.2, and stack tool version 2.3.1. See https://docs.haskellstack.org/en/stable/README/#how-to-install for a manual installation.

Artifact Contents

We provide the source code of all of our examples. Additionally, the docker image contains the `vim` editor with Haskell support, for inspecting the files. The main files in the artifact are:

- `README.md` : This file
- `LICENSE` : BSD-3 license
- `src` : Source code of the cost-aware multiparty protocol library
- `app` : Script used for estimating sequential costs from benchmarking data
- `examples` : Cost-aware protocols used in the *CAMP* paper

- `benchmarks` : Real execution times used in the comparison against the cost models.
- `run.sh` : Script to gather the costs of all the examples, using the current input parameters.

Executing the Artifact

We refer to file `examples/CParallel.hs` for a quick test of the artifact.

Overview of `CParallel.hs` : This file contains a number of cost-annotated protocols, as well as real execution times taken from directory `benchmarks` . Definitions with type `GTM a` are monadic computations that specify a (part of) a protocol. For example,

```
fftTopo :: String -> [Role] -> GTM ()
```

defines a butterfly topology in a divide-and-conquer manner using the list of participants (type `[Role]`). Function `gclose` runs computations of type `GTM a` , and produces the global type that they represent. For example, function

```
mkFft :: Int -> CGT
```

generates a butterfly topology of size `n :: Int` . From these cost-aware global types, we produce cost equations (function `cost`), and instantiate the parameters (function `evalTime`).

Evaluating cost equations: to evaluate these cost equations, move to the initial working directory, and run the file using command `stack` (**note**, the command may take a few seconds to complete):

```
$ cd /home/oopsla20-artifact/CAMP
$ stack runhaskell examples/CParallel.hs
%% FFT: & cost & measured & error
& 143.10000294
& 143.016404
& 5.845409174181284e-4

& 74.30000588
& 74.175541
& 1.6779773807110488e-3

& 40.52500882
& 40.817588
& 7.167968376769449e-3

& 24.26251176
& 21.755833
& 0.11521869836011335

& 16.7562647
& 14.519975
& 0.1540147073255979

& 13.62814264
& 12.467197
& 9.312002048255l1e-2

& 12.68908308
& 11.970078
& 6.0066866732196626e-2

& 12.84455183
& 12.686601
& 1.245020868867877e-2



%% MS: & cost & measured & error
& 98.145231
& 98.145231
& 0.0

& 53.57262789
& 53.183744
& 7.312081864714197e-3
```

```
& 31.28633686999999
& 31.325671
& 1.2556516347251097e-3

& 20.143208404999978
& 18.091493
& 0.11340774390482744

& 14.571674237500002
& 14.214225
& 2.514728995073603e-2
```

The output represents the protocol costs, the real execution times, and the error (computed as `abs (measured - cost) / measured`). The measured execution times are hardcoded, and are taken from `benchmarks/c-pthread/benchmarks/FFT/data/t_12`, and `benchmarks/c-pthread/benchmarks/Mergesort/data/t_12`.

Part 2: Step by Step Instructions

camp is the implementation of the cost-aware multiparty protocols described in *CAMP: Cost-Aware Multiparty Session Protocols*. The implementation is a Haskell library, that provides a monadic interface for specifying multiparty protocols.

To evaluate the cost models for all of our cost-aware protocols, we provide script `./run.sh`.

Paper claims supported by this artifact: The paper, in last paragraph of §1 (p. 3) states "The anonymised git repository https://github.com/camp-cost/camp provides a working prototype implementation, described in §7 and the data used in §8, with instructions for replicating our experiments. We will also submit it as an artifact.". The main claims that this artifact supports are thus those ones related to §7 and §8:

- *we can extract cost equations from global types*
- *we can predict the throughput of recursive protocols*
- *the cost models address asynchronous message optimisations*
- *cost-aware global types can estimate real execution costs*

We will walk through these claims in these step-by-step instructions.

Not expected to be supported by this artifact (and the paper):

- **performance**: the benchmarks are taken from multiple sources, each with their on execution environments. We do not address replicating their reported execution times, but establishing their cost-aware protocols. Running their benchmarks in a docker container or virtual machine may affect the execution times, and this docker image is not prepared to address replicating their experiments. Accurate predictions may require extensive profiling to understand the different execution times of the different parts of the protocols, which is also out of the scope of this artifact.

To evaluate these claims, we propose to: (1) inspect the protocol definitions under `examples`; (2) inspect the benchmarking data under `benchmarks`; and (3) run the cost estimation scripts under `examples`. Before walking through these examples, we will provide an overview of our current prototype implementation.

Overview of the Implementation

The source code is under directory `src`, and the main modules are:

- `Language.SessionTypes.Common`: common definitions for Global and Local Types.
- `Language.SessionTypes.Global`: global types.
- `Language.SessionTypes.Local`: local types.
- `Language.SessionTypes.Cost`: cost-aware global types.
- `Language.SessionTypes.Seq`: functions to instantiate sequential costs from benchmarking data.

Cost-Aware Protocol Definitions

Directory *examples* contains the cost-aware protocol definitions used in the *CAMP* paper. For example, consider the following definition in **OCamlGT.hs**.

```
rpingpong :: CGT
rpingpong = gclose $ do
  p <- mkRole
  q <- mkRole
  grec 1 $ \x -> do
    message p q (Var "\\tau_1") (CVar "c_1")
    message q p (Var "\\tau_2") (CVar "c_2")
    x
```

In this definition, `CGT` is the type of cost-aware global types. Function `gclose :: GTM () -> CGT` takes a global type script (type `GTM ()`), and produces a cost-aware global type. A global type script is a monadic computation that describes a protocol. Action `mkRole` creates a new participant in the protocol, `grec n $ \x -> do G` specifies a recursive protocol `G`, where `x` is the recursion variable, and `n` is the annotation the amount of iterations that the protocol is expected to run on average.

Variables `\\tau_1` and `c_1` represent type and cost variables, that will occur in the cost expressions extracted from this protocol. To extract cost equations, we use functions:

```
throughput :: CGT -> Time
cost :: CGT -> Time
```

Cost equations can be instatiated by providing: (a) instantiations for the communication costs, and (b) instantiation of cost and size variables. By instantiating the equations, we obtain a mapping from participants to execution times. We provide function `evalTime` for this purpose:

```
evalTime :: Map (Role, Role) (Double -> Double)
         -> Map (Role, Role) (Double -> Double)
         -> Map String Double
         -> Time
         -> Map Role Double
```

The full list of protocol specifications is described in §8.2 (p.18). They can be found in the following files in our artifact:

OCaml benchmarks by Imai et al. [2020]: This is a recursive ping-pong, with three different transports, `ev`, `lwt` and `ipc`. The protocols can be found under `examples/OCamlGT.hs`.

Go benchmarks by Castro et al. [2019]: The examples are in `examples/ScribbleGo.hs`. The benchmarks are divided in two categories:

(1) microbenchmarks, `aa`, `a1` and `1a` (functions `goAA`, `go1A` and `goA1` respectively)

(2) CLBG benchmarks, `sn`, `kn` and `dna` (functions `snP`, `knucP` and `dnaP` respectively).

Savina benchmarking suite. They can be found under directory `examples/savina`. They are separated into `A.microbenchmarks` and `B.conc`. The files are given the same names as in the paper (See last paragraph of p.18, and paragraph *Concurrency benchmarks* in p.19).

C-MPI by Ng et al. [2019]. These are in file `examples/PabbleMPI.hs`. Functions `ringP`, `meshP`, `scP` capture ring, mesh and scatter-gather topologies, used by `nb`, `ls`, and `wc` and `ap` respectively (see Fig. 8).

C-pthreads. File `examples/CParallel.hs`. Functions `mkFft` and `dcP` are the butterfly and divide-and-conquer topologies for the FFT and Mergesort benchmarks.

Two-buyer and Double buffering. These are under `examples/TwoBuy.hs` and `exampels/DBuff.hs`.

Benchmarks

The benchmarking data and code (or link to the original source code) used in this paper can be found in directory *benchmarks*. Each benchmark has a different structure. For example, directory `c-pthread/benchmarks/FFT/data/t_12` contains the benchmarking data for running `FFT` on a 12-core machine. The file has the following structure:

```
...
size: 4194304
```

```
        K: seq
                mean: 7.182371
                stddev: 0.048188
        K: 0
                mean: 7.283062
                stddev: 0.097863
        K: 1
                mean: 7.250648
                stddev: 0.116543
        K: 2
                mean: 3.503649
                stddev: 0.042035
        K: 3
                mean: 1.752671
                stddev: 0.039644
        K: 4
                mean: 0.944878
                stddev: 0.018909
        K: 5
                mean: 0.673194
                stddev: 0.017714
        K: 6
                mean: 0.594781
                stddev: 0.010795
        K: 7
                mean: 0.621905
                stddev: 0.009588
        K: 8
                mean: 0.717387
                stddev: 0.015388
...
```

This file gathers the real execution times for FFT on a 12-core machine. Each entry contains the input size (e.g. 4194304), and the measured execution times with different butterfly topology sizes (2^ `K` , for the cases where `K` is not `seq`), as well as the sequential costs (case `K: seq`).

These real costs are compared against the costs predicted from the cost-aware protocols.

Running Cost Estimations

To run the cost estimations, and print the results from instantiating the cost equations, please run

```
$ stack runhaskell <example>.hs
```

on each Haskell file under `examples` . For example, executing the command

```
$ stack runhaskell examples/CParallel.hs
```

will produce the following result:

```
%% FFT: & cost & measured & error
& 143.10000294
& 143.016404
& 5.845409174181284e-4

& 74.30000588
& 74.175541
& 1.6779773807110488e-3

& 40.52500882
& 40.817588
& 7.167968376769449e-3

& 24.26251176
& 21.755833
& 0.11521869836011335

& 16.7562647
& 14.519975
```

```
& 0.1540147073255979

& 13.62814264
& 12.467197
& 9.31200204825511e-2

& 12.68908308
& 11.970078
& 6.0066866732196626e-2

& 12.84455183
& 12.686601
& 1.245020868867877e-2

...
```

Finally, it is possible to evaluate these examples in GHCi. For example:

```
$ stack ghci examples/CParallel.hs
```

Then, once in GHCi, you can browse the definitions of the module:

```
*CParallel> :browse
mkRoles :: Int -> GTM [Role]
...
fft :: [Double]
```

Additionally, you can print global types

```
*CParallel> print $ mkFft 1
CGSend 0 1 (Var "\\tau") (CGSend 1 0 (Var "\\tau") (CGRecv 0 1 (Var "\\tau")
(CVar "c_1") (CGRecv 1 0 (Var "\\tau") (CVar "c_1") CGEnd)))
```

and print the generated cost equations:

```
*CParallel> putStrLn $ showEqns $ cost $ mkFft 1
r0 = T_recv (r0, r1, \tau) + c_1 + max (T_send (r0, r1, \tau), T_send (r1, r0, \tau))
r1 = T_recv (r1, r0, \tau) + c_1 + max (T_send (r1, r0, \tau), T_send (r0, r1, \tau))
```