Formal Foundations of Serverless Computing

Abhinav Jangda, Donald Pickney, Samuel Baxter, Joseph Spitzer, Breanna Devore-McDonald, Yuriy Brun, Arjun Guha
CAT
Serverless Platforms

Lambda

Google Cloud Platform

Azure
Supported Languages

- Node.js
- Python
- Java
- Swift
- PHP
```javascript
model = CatOrDogModel ();

function catOrDog (req, res) {
    if (req.type === 'train') {
        model.train (req.data);
    } else {
        res.write (model.predict (req.data));
    }
}
```

- Store trained model
- Takes a HTTP Req
- Train Model
- Predict Cat or Dog
```javascript
model = CatOrDogModel();
function catOrDog (req, res) {
  if (req.type === 'train') {
    model.train (req.data);
  } else {
    res.write (model.predict (req.data));
  }
}
```
```javascript
model = CatOrDogModel();
function catOrDog (req, res) {
    if (req == 'train') {
        model.train (req.data);
    } else {
        res.write (model.predict (req.data));
    }
}
```
model = CatOrDogModel ();
function catOrDog (req, res) {
  if (req.type === 'train') {
    model.train (req.data);
  } else {
    res.write (model.predict (req.data));
  }

More than one instances of function can be running in parallel.
let Datastore = require('@google-cloud/datastore');

function catOrDog(req, res) {
    let ds = new Datastore({ projectId: 'cat-app' });
    let dst = ds.transaction();
    dst.run(function() {
        let tId = ds.key(['Transaction', req.body.transId]);
        dst.get(tId, function(err, trans) {
            if (err || trans) {
                dst.rollback(function() { res.send(false); });
            } else if (req.body.type === 'train') {
                let data = ds.key(['model', req.body.data]);
                dst.get(data, function(err, model) {
                    model.train(data)
                    dst.save({ key: data, data: model });
                    dst.save({ key: tId, data: {} });
                    dst.commit(function() { res.send(true); });
                });
            } else if (req == 'test') {
                let data = ds.key(['prediction', req.body.data]);
                dst.get(data, function(err, model) {
                    prediction = model.predict(data)
                    dst.save({ key: data, data: prediction });
                    dst.save({ key: tId, data: {} });
                    dst.commit(function() { res.send(prediction); });
                });
            } else {
                dst.rollback(function(){res.send(false);});
            }
        });
    });
}

model = CatOrDogModel();

function catOrDog(req, res) {
    if (req.type == 'train') {
        model.train(req.data);
    } else {
        res.write(model.predict(req.data));
    }
}

= x 4
Serverless Programming is Hard

- Challenges
  - Ephemeral State
  - Need for transactions
  - Concurrency
- Need a deep understanding of serverless platforms to provide robust tools to programmers for writing correct code.

“Serverless functions are not functions in an ordinary sense”
Our Contributions

- Operational semantics to model essential details of serverless platforms.
- Three case studies to show that the semantics is useful:
  - Idealized Semantics
  - Key Value Store
  - Serverless Programming Language for composing functions
Operational Semantics of Serverless System

\[
\text{REQ} \quad \frac{x \text{ is fresh}}{C \xrightarrow{\text{start}(x, v)} CR(f, x, v)}
\]

\[
\text{COLD} \quad \frac{y \text{ is fresh} \quad \text{recv}(v, \sigma_0) = \sigma}{CR(f, x, v) \Rightarrow CR(f, x, v)E(f, \text{busy}(x), \sigma, y)}
\]

\[
\text{WARM} \quad \frac{\text{recv}(v, \sigma) = \sigma'}{CR(f, x, v)E(f, \text{idle}, \sigma, y) \Rightarrow CR(f, x, v)E(f, \text{busy}(x), \sigma', y)}
\]

\[
\text{HIDDEN} \quad \frac{\text{step}(\sigma) = (\sigma', \varepsilon)}{CE(f, \text{busy}(x), \sigma, y) \Rightarrow CE(f, \text{busy}(x), \sigma', y)}
\]

\[
\text{RESP} \quad \frac{\text{step}(\sigma) = (\sigma', \text{return}(v'))}{CR(f, x, v)E(f, \text{busy}(x), \sigma, y) \xrightarrow{\text{stop}(x, v')} CE(f, \text{idle}, \sigma, y)S(x, v')}
\]

\[
\text{DIE} \quad \frac{CE(f, m, \sigma, y) \Rightarrow C}{}
\]
Operational Semantics of Serverless System
Operational Semantics of Serverless System

- **COLD-START**
  - New NodeJS
  - Idle NodeJS

- **WARM-START**
  - Idle NodeJS
  - New NodeJS
  - Idle NodeJS processing request
Summary

Operational semantics models all details of serverless platforms:
1. New instance creation
2. Instance reuse
3. Ephemeral and persistent state

Ideally: Provide programmers higher-level abstractions
1\textsuperscript{st} Case Study: Idealized Serverless Semantics

Models an idealized serverless platform with:
1\textsuperscript{st} Case Study: Idealized Serverless Semantics

\[
\begin{align*}
N\text{-}START & \quad \frac{x \text{ is fresh} \quad \text{recv}(v, \sigma_0) = \sigma'}{\langle f, \text{idle}, \vec{\sigma} \rangle \xrightarrow{\text{start}(x, v)} \langle f, \text{busy}(x), [\sigma_0, \sigma'] \rangle} \\
N\text{-}STEP & \quad \frac{\text{step}(\sigma) = (\sigma', \varepsilon)}{\langle f, \text{busy}(x), \vec{\sigma} + [\sigma] \rangle \rightarrow \langle f, \text{busy}(x), \vec{\sigma} + [\sigma, \sigma'] \rangle} \\
N\text{-}STOP & \quad \frac{\text{step}(\sigma) = (\sigma', \text{return}(v))}{\langle f, \text{busy}(x), \vec{\sigma} + [\sigma] \rangle \xrightarrow{\text{stop}(x, v)} \langle f, \text{idle}, [\sigma] \rangle}
\end{align*}
\]
Theorem: If certain conditions are met, then Idealized Semantics is weakly bisimilar to Operational Semantics.
2\textsuperscript{nd} Case Study: Semantics with Key Value Store
3rd Case Study: Serverless Programming Language

1. Extend Operational Semantics with a serverless composition language
2. Inspired by OpenWhisk Conductor and IBM Composer
Serverless Compositions

BlackOrWhite

Black
Serverless Compositions

CatOrDog

BlackOrWhite

Black Cat
3rd Case Study: Serverless Programming Language

```
a <- invoke catOrDog (in);
return invoke blackOrWhite (a.animal);
```
1. OpenWhisk Conductor is executed in a docker container.
2. SPL’s JSON Transformation can be executed directly in OpenWhisk Controller.
Contributions

1. Writing correct code for serverless platform is hard
2. We present Operational Semantics of serverless platform
3. We present 3 case studies to show these semantics are useful
   a. Naive Semantics are abstractions of operational semantics
   b. Operational Semantics are extended with a Key Value store
   c. Serverless Programming Language can be used to compose existing serverless functions efficiently.
1st Case Study: Naive Serverless Semantics

Theorem: If certain conditions are met, then Naive Semantics is weakly bisimilar to Operational Semantics.
3rd Case Study: Serverless Compositions

Sufficient to describe control flow
Issues

```javascript
let accounts = new Map();
exports.bank = function(req, res) {
    accounts.set(req.body.name, req.body.value);
    res.send(true);
};
```

➢ Ephemeral State
➢ More than one instances of function can be running in parallel.
➢ Re-invoke functions when a failure is detected.
   A single event may be processed to completion multiple times.
Operational Semantics of Serverless System

➢ Based on experiences with major serverless computing platforms.
Operational Semantics of Serverless System

A new Function instance of \( f \) with initial state \( \sigma \) for \( id \) has started

Start executing \( f \) on an \emph{idle} function instance
Operational Semantics of Serverless System

\[ F(f, m, \sigma) \] 

DIE
Introduction to Serverless Computing

Deploys JavaScript function, $foo$ on the cloud server

Calls function $foo$ on the server

Receives data after executing function $foo$
- Focus on application code only
- Compiles code
- Configures OS
- Manages resource allocation.
Serverless Function Example

```javascript
let accounts = new Map();
exports.bank = function(req, res) {
    accounts.set(req.body.name, req.body.value);
    res.send(true);
};
```

- Store account data
- Takes a HTTP Req
- Deposit the amount
Serverless Programming Language

```
a <- invoke f (in);
return invoke g (a.x);
```

- **in** is the input to the composition
- First element of array as the input to the composition
- Perform JSON Transformations to store the state of composition
Serverless Programming Language

\[
a \leftarrow \text{invoke } f \ (\text{in}) ; \\
\text{return } \text{invoke } g \ (a . x) ;
\]
Operational Semantics of Serverless System

\[ + R(f, id, v) \]
\[ + F(f, busy(id), \sigma) \]

RESPOND

\[ + F(f, idle, \sigma) \]

Function instance has turned *idle*. 
Operational Semantics of Serverless System

\[ + F(f, \text{idle}, \sigma) \xrightarrow{S(\text{result})} \text{RESPOND} \]
Operational Semantics of Serverless System

\[ \text{WARM-START} \]

Start executing \( f \) on an \( idle \) function instance
Need for Serverless Compositions

Scheduler spends most of the time idle. Leading to double billing
Need for Serverless Compositions

Merge f and g into one function

1. Hinders code-reuse
2. Does not work when
   a. source code is unavailable or
   b. the serverless functions are written in different languages
Need for Serverless Compositions

The solution should

- not lead to double billing
- does not hinder code-reuse
- be language agnostic
Serverless Programming Language

Composition primitives for serverless platforms based on Haskell Arrows

invoke

f >>> g

Sufficient to describe control flow
Serverless Programming Language

Implementations:

1. Integrated in a fork of OpenWhisk
2. Portable implementation to communicate with other public cloud providers
Serverless Programming Language

1. Comparison to OpenWhisk Conductor
2. Increasing Request Sizes
3. 6-core Intel Xeon E5-1650 with 64 GB RAM with Hyper-Threading enabled.