ohua Documentation

Release 0.6.8

Sebastian Ertel, Justus Adam

Dec 12, 2018

Contents

Contents:

CHAPTER 1

Stateful functions

A stateful function is a piece of 'native' code which is sequentially executed by ohua and has for its execution context some associated, opaque state.

Concretely this means that a stateful function, in Java for instance, is a method and an associated class.

An example:

```java
public class IntegerAccumulator {
    private int counter = 0;

    @defsfn
    public int increment(int valueToAdd) {
        counter += valueToAdd;

        return counter;
    }
}
```

For each invokation site of the stateful function ohua creates a new instance of the associated class. And each time that particular piece of code is run the same instance of the associated class is handed to the method.

Defining stateful functions is currently supported for three languages: *Java*, *Clojure* and *Scala*. In theory however any language can define a stateful function, so long as it creates JVM bytecode for a class and a method.

1.1 Stateful functions in Java

Defining stateful functions in Java is very simple. Every stateful function needs an associated class.

The class must satisfy the following conditions

1. **The class must be `public`.** This enables the runtime to find it.
2. **The class has a default constructor and is not `abstract`.** Since the runtime cannot know what the constructor arguments should be it will attempt to instantiate it with none.

3. **Only one stateful funciton is defined on the class.** The class may define as many methods and members as it wants, however it may only define one stateful function. This restriction may be lifted in the future. However for the present if you wish to define multiple stateful functions in one file we suggest using static inner classes.
4. **If the class is an inner class it must be `static`.** Otherwise the runtime will be unable to instantiate the class.

To define the stateful function on the class itself simply annotate the desired method with `@defsfn`.

```java
public class AssociatedClass {

    @defsfn
    public String concat(String a, String b) {
        return a + b;
    }
}
```

The method must satisfy the following conditions

1. **The method must be `public`.** Otherwise the runtime will not be able to call it.
2. **The method must not be `static`.** Using a static method will lead to an arity exception. If you must have a static version of your code define the static method and then define a second, non static method which calls the static one and annotate this one with `defsfn`.

Stateful functions defined in Java may be brought into scope for use in an algo using *Bringing stateful functions into scope*.

1.2 Stateful functions in Clojure

Stateful functions defined in Clojure may be brought into scope for use in an algo using standard Clojure `require`.

1.2.1 Stateless Clojure functions

You can use any normal clojure function in ohua. User defined functions as well as library functions can be directly called in the ohua EDSL.

Warning: As of yet there is no support for lambdas

As an example, this does not work:

```clojure
(ohua
  (let [x (accept socket)
        lam (fn [y] ( ... x))]
    ..)
```

1.2.2 Stateful Clojure functions

Stateful functions in Clojure are simply Clojure functions, which have been annotated with the metadata `:init-state`. This `:init-state` metadata contains a Clojure expression which initializes the state for the stateful function. This can be any Clojure expression and it may produce any Clojure data structure. The *exact reference* returned by `:init-state` will be passed to every invokation of the stateful function. Since this state reference will be passed to the function when invoked every Clojure stateful function must have as its first argument the reference for the state, usually called `this`. Therefore if you require mutable state, we recommend using clojure atoms, mutable Java data structures or mutable clojure data structures.

There is a convenience macro called `defsfn` which works like `defn` but additionally takes as a second argument the `:init-state` expression.

```
; defined with defsfn
(defsfn aggregate (new java.util.ArrayList) [this arg1]
  (if (> (.size this) 6)
    (let [copy (new java.util.ArrayList this)]
      ; mutable actions are allowed
      (.clear this)
      copy)
    (.add this arg1)))

; defined with defn
(defn ^{:inti-state '(atom #{})} was-seen [this thing]
  (if (contains? @this thing)
    true
    (do
      (swap this conj thing)
      false)))
```

1.3 Stateful functions in Scala

Defnining stateful functions in Scala is basically identical to defining stateful functions in Java. See the requirements for the method and associated class in *How to define stateful functions in Java*. Annotate the method with `@defsfn`.

```
class Concat {
    @defsfn
    def concat(a:String, b:String) -> String = {
        a + b
    }
}
```

Stateful functions defined in Java may be brought into scope for use in an algo using *Bringing stateful functions into scope*.

CHAPTER 2

Importing syntax

Stateful functions for use in the `ohua` macro are being brought into scope with a macro called `com.ohua.lang/ohua-require`. The `ohua-require` macro scans the classpath for the specified functions, loads them and makes them available for use with `ohua` in the namespace in which it was called. It supports qualified and unqualified imports as well as aliasing for both namespaces/packages and functions.

Important: Importing stateful functions like this makes them available *only* for use in the `ohua` macro. They will not be available via the standard clojure symbol resolution. If you *must* resolve the functions yourself use `com.ohua.link/resolve`.

The syntax for `ohua-require` is intentionally similar to `clojure.core/require`.

```
(ohua-require
    [my.package]
    [my.package :as package]
    [my.package :refer [function]]
    [my.package :refer :all])
```

The macro takes a lists or vectors as input. The first element of each is the name of a package/namespace which should be imported. Functions from the package are then available qualified as `my.package/function`. Optionally you may alias the package using the `:as` keyword and providing a new name to bind the package to. And lastly you may use `:refer` to specify functions which should be imported such that they may be used unqualified. `:refer :all` makes all functions from the package available as unqualified imports. These are the only `require` semantics which are currently implemented.

CHAPTER 3

The standalone compiler: `ohuac`

The `ohuac` executable is a standalone program which can compile a source file written in one of the ohua *Frontends for ohua algorithms* and *generate code* for various platforms.

3.1 Getting it

Prebuilt binaries are available for 64bit Linux and OSX. Provided for each release by Travis CI.

Simply download the appropriate executable for you platform (`ohuac-linux` or `ohuac-osx`) from the releases page.

3.1.1 Building from source / Getting the source code

The source code for the compiler can be found on GitHub. However major parts of the compiler are separate libraries. For more information on the repository structure see the *project organisation page*.

`ohuac` is written in Haskell and uses the stack tool for building and dependency management. If you have stack installed already you can simply download the source from the releases page and run `stack install`.

This downloads and builds all dependencies, as well as the Haskell compiler `ghc`, should it not be present already. It should not interfere with any system-level libraries, Haskell ecosystems etc.

It builds the executable `ohuac` and copies it to `~/.local/bin`. If you do not wish this use `stack build` instead and find the path of the finished binary using `stack exec -- which ohuac` afterwards. After building the binary can be freely moved to any location on the system.

3.2 Usage

The capabilities and options for the compiler can be interactively explored by calling `ohuac --help` or `ohuac COMMAND --help` to get help for a specific `COMMAND`.

3.2.1 The `build` command

The most common command is `build`.

The build command transitively builds your ohua modules. This means it not only compiles the one module you directly specify but also all modules it may depend on.

Currently the search path for modules cannot be influenced. Therefore they must be in a specific location, which is `module/submodule.ohuac`. The type of file (`.ohuac` or `.ohuas`) does not matter in this case. In fact they can also be mixed. A `ohuas` file can import a module that is defined as a `ohuac` file.

As an example we may have a module `A.ohuac`, which depends on `foo.bar.B` and `foo.C`.

```
A
|-- foo.bar.B
'-- foo.C
```

During compilation the compiler will look for these dependencies at `foo/bar/B.ohuac` or `foo/bar/B.ohuas` and `foo/C.ohuac` or `foo/C.ohuas`. All paths being relative to the current working directory. So if the compiler was called with `ohuac build A.ohuac`, the working directory is expected to look something like this

```
WORKING_DIR/
     |-- A.ohuac
     '-- foo/
          |--bar/
          |   '-- B.ohuac
          '-- C.ohuas
```

If both an `.ohuac` and `.ohuas` file is found for a particular dependency the compiler will exit with an error.

3.2.2 Universal options

The options listed in `ohuac --help` apply to all subcommands. They are generally **prepended** to the options passed to the subcommand.

3.3 Supported targets for code generation

The standalone compiler currently supports two compilation targets. One is a universal format, which simply encodes the dataflow graph directly in JSON and a second which creates a java class.

3.3.1 JSON Graph repesentation

The JSON output from `ohua` is selected with the `-g json-graph` option. The default file extension is `.ohuao`.

This representation only encodes the dataflow graph and the stateful functions used. It is intended to be a universal, easily to use intermediate for backeds which have no direct support in `ohuac` yet. Such a backend only has to define the runtime for the dataflow execution and the loading and linking of stateful functions, `ohuac` will take care of parsing, interpreting and optimising the algorithms.

3.3.2 A simple runnable java class

The `-g simple-java-class` code gen option is a very lightweight code generation for the java platform.

It generates a class where the module path is its package and the algorithm name is the name of the class. E.g. namespace `foo.bar.baz` with the `main` algorithm selected generates a class `foo.bar.baz.Main`.

For an algorithm `main` with arguments `A`, `B`, `C` and return type of `D`, the class has the following structure

```
class Main {
   public static C invoke(A argument0, B argument1, C argument2);
   public static Configured configure(RuntimeProcessConfiguration configuration);

   public static class Configured {
       public Prepared prepare(A argument0, B argument1, C argument2);
   }
   public static class Prepared {
       public static D invoke();
   }
}
```

The argument and return types default to `Object`.

The structure of the class follows a simple schema. Each algorithm class has two `static` methods `invoke` and `configure`. `invoke` simply executes the algorithm with the default runtime parameters. `configure` allows customization of runtime parameters.

Furthermore there are always two inner classes, `Configured` and `Prepared`. The two inner classes represent stages of algorithm configuration. `Configured` is a graph with an associated runtime configuration and can be turned into a `Prepared` by calling `prepare` with the arguments specified in the algorithm description.

CHAPTER 4

Frontends for ohua algorithms

CHAPTER 5

Algorithms

Algorithms are ohua's high level abstraction of a dataflow graph.

An algorithm combines stateful functions and other algorithms into a program which can then be executed by the `ohua` macro. The `ohua` macro decomposes the algo into a dataflow graph which it executes in parallel.

5.1 Defining algorithms

The `com.ohua.lang/algo` macro allows one to define an algo using the Clojure programming language. All functions which are used in the algo must be in scope in the current namespace. See *Bringing stateful functions into scope* for more information.

5.2 Bringing stateful functions into scope

5.3 Runtime execution model

CHAPTER 6

Organisation of the ohua project

6.1 Repositories

ohua-core https://github.com/ohua-dev/ohua-core

This is the heart of the compiler. The core library that defines the intermediate language, the dataflow graph, transitions, optimisations, hooks etc.

ohuac https://github.com/ohua-dev/ohuac

The standalone compiler. An executable which combines ohua *Frontends for ohua algorithms* and core to parse algorithm files and generate code for various target platforms.

CHAPTER 7

Documentation of compiler internals and API

7.1 The dataflow language `DFLang`

The dataflow language is the bridge between the call by need lambda calculus and a dataflow graph. It retains aspects of both but is neither completely. It is represented as a sequence of `let` assignments, where the right hand side of the assignment is always a call of a stateful function or dataflow operator. Multiple arguments can be given to this call, however they must always be references to either a locally (`let`) bound value, or an environment reference.

It is thus much more restricted than the algorithm language because it does not support lambdas, and the RHS of `let` is not an arbitrary expression.

The dataflow language is defined in `Ohua.DFLang.Lang`.

7.1.1 Writing `DFLang`

It is sometime necessary (in particular for tests) to create `DFLang` values. To avoid having to write them directly in Haskell, the `ohua-test-tools` library includes a `DFLang` parser. This parser can be used in Haskell code when the `QuasiQuotes` extension is enabled by importing `Ohua.Test` and using the `[embedDflang| Your dflang expression |]` quasi quoter.

It supports the same style comments as the algorithm language.

An example:

```
let (x) (* target bindings *)
            = some.package/sf (* function reference *)
              <0> (* call site id *)
              [s] (* state source (optional) *)
              (a, b, c) (* arguments *) in
let (y, z) = dataflow other.package/a_dataflow_fn<1>() in
x
```

Target Bindings A tuple of bindings that are created by this function call. The names should be unique in the expression. Always has to be a tuple, i.e. comma separated and surrounded by parentheses, even if only one value is created. Can be empty.

Function Reference The fully qualified reference to the function being called. The syntax is `namespace/functionName`, i.e. `ohua.lang/id`. May optionally be preceded by the keyword `dataflow` to indicate that this is not a simple function but a dataflow operator, i.e. translates to the `nodeType` being `OperatorNode`.

Call Site Id A positive integer uniquely identifying this call. Uniqueness is not checked but the behavior of the compiler is undefined if call site id's are not unique.

State Source A value to be used as initial state for this function. Must be specified for stateful functions and omitted for stateless functions. Can be a literal of binding (same as *arguments*).

Arguments A tuple of arguments as input to the function. As with *target bindings* these need to be comma separated and surrounded by parentheses, but can be empty. In addition to local variables, such as `a`, `b`, `myVar`, numeric literals `1`, `0`, `-4`, the unit literal `()` and references to environment expressions are allowed as arguments. The latter are positive numbers prefixed with `$`, i.e. `$1`, `$100`.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search