Algorithm Engineering with Clojure

Gunnar Völkel
University of Ulm

01.05.2012

joint work with Johann M. Kraus and Hans A. Kestler
method for algorithmic research

cycle of *algorithm design*, *analysis*, *implementation* and *experimental evaluation*

our goal: assisting algorithm implementation and experimental evaluation
Clojure

- new Lisp dialect on the Java Virtual Machine
- strong functional orientation
- general-purpose programming language

- our reasons to choose Clojure:
  - immutable data structures – concurrency promises
  - first-class functions: facilitate metaheuristic implementation
  - usage of existing Java libraries
Algorithm implementation:
- error search: tracing of functions
- performance optimization: timing of functions

Experimental evaluation:
- compact experiment setup specification via DSL
  - experiment definition
  - parameter selection
  - problem instance selection
- automatic execution of experiments
  - creation of individual experiments from setup
  - execution with progress report and remaining runtime estimation
2 Development Tools

- Intercepting Function Definitions
- Tracing Functions
- Timing Functions
Intercepting Function Definitions

- Clojure’s main building blocks: functions and namespaces
  
  ```clojure
  (ns example)
  
  (defn perfect?
    "Determines whether n is a perfect number."
    [n]
    (= n (reduce + (filter #(= 0 (mod n %)) (range 1 n)))))
  ```

- intercepting Clojure function definitions for implementation of development tools

- interception example:
  
  ```clojure
  (defn perfect?
    "Determines whether n is a perfect number."
    [n]
    (let [result
      (= n
        (reduce + (filter #(= 0 (mod n %)) (range 1 n))))]
      (println "(perfect?" n ") =" result)
      result))
  ```
Roadmap: Interception

- per function
  - defn $\Rightarrow$ defn-intercept
- per namespace
  - intercept-setup
- global
  - external configuration
Interception Macro: **defn-intercept**

- **assumption**: fixed interception function
  ```clj
  (defn print-interception [fsymb, params, body]
    (println
      (format "Defining: (%s %s %s)" fsymb params body))
    (list params body))
  )

- **function interception implementation**:  
  ```clj
  (defmacro defn-intercept [fsymb, params, body]
    (let [modified (print-interception fsymb, params, body)]
      `(defn ~fsymb ~@modified)))
  )

- **usage example**:  
  ```clj
  (defn-intercept perfect? [n]
    (= n (reduce + (filter #(= 0 (mod n %)) (range 1 n)))))
  )

  ; "Defining: (perfect? [n] (= n (reduce + (filter #(= 0 (mod n %)) (range 1 n))))"
  ;=> #'example/perfect?
  ```
• inconvenient: replacing `defn` with `defn-intercept`  

• strong limitation: fixed interception function  
  
  (two development tools: tracing and timing)
registration of interception function and definition of a setup macro, e.g.

(ns trace)
(defn trace-interception [fns, fsymb, params-body] ...)
(create-setup trace-setup, trace-interception)

⇒ macro trace/trace-setup

usage example:

(ns example (:use trace))
(trace-setup perfect?)

(defn perfect? [n] <impl>)
(defn primes [n] <impl>)

(ns example (:use trace))
(trace-setup perfect?)

(defn perfect? [n] (trace-interception <impl>))
(defn primes [n] <impl>)
creation of interception setup macro:

\[(\text{create\-setup } \text{trace\-setup}, \text{trace\-interception})\]

setup macro registers function names for interception

\[(\text{trace\-setup } \text{perfect?}, \text{sum})\]
\[(\text{timing\-setup } \text{sum}, \text{square})\]

Registry

- \text{perfect?}
- \text{sum}
- \text{square}
- \ldots

- \text{trace\-interception}
- \text{trace\-interception, timing\-interception}
- \text{timing\-interception}

revised \text{defn\-intercept} using registry

setup macro replaces \text{defn} with \text{defn\-intercept} in current namespace
• **convenient**: no `defn` replacement

• **unlimited**: arbitrary interception functions – more than one per function possible

• **still inconvenient**: need to add setup macro call to the source file
Global Interception Setup

- external configuration file for function interception:
  
  ```lisp
  (enable-intercept true)
  (trace example/perfect? example/sum)
  (timing example/square)
  ```

- `trace`, `timing` associated with an interception function

- **bootstrap code** needed to replace Clojure’s `defn`

  - **bootstrap** with **Leiningen** (Clojure build tool) via `project.clj`

    ```lisp
    (defproject Examples 0.1
      ...
      :project-init
      (do (use 'debug.intercept)
          (setup-global-interception "config.clj"
           '[trace/trace, timing/timing])))
    ```

- not limited to Leiningen
macro setup-global-interception replaces defn with defn-intercept globally

(alter-var-root #'clojure.core/defn (constantly (var-get #'defn-intercept)))

extendable via additional configuration commands, e.g. trace from namespace trace

(setup-global-interception "config.clj" '[trace/trace, timing/timing]))

different interception on every jar execution provided that
  only namespace with main method is ahead-of-time compiled
  other namespaces as plain clojure source
  setup-global-interception called in main method
  ("runtime-config.clj")
- **transparent**: no source file changes
- **unlimited**: arbitrary interception functions
- runtime interception configuration for standalone jars
given a naïve fibonacci implementation:

```
(defn fib [n]
  (cond
    (< n 0) 0,
    (< n 2) 1,
    :else (+ (fib (- n 1)) (fib (- n 2)))))
```

trace tree for `(fib 4)`:

```
(fib 4)
  5
  /  
(fib 3)  (fib 2)
  3    2
 /   /   /
(fib 2) (fib 1) (fib 1) (fib 0)
  2    1    1   1
```

```
(fib 4)
  5
  /  
(fib 3)  (fib 2)
  3    2
 /   /   /
(fib 2) (fib 1) (fib 1) (fib 0)
  2    1    1   1
```
Trace Tree

- nodes for **direct** and **indirect calls** to the traced functions
- nodes for the **executed forms** within the traced functions

Trace Node Content

- function information
- parameters used for this invocation
- return value of this invocation
- raised exception of this invocation (if any)
(defn fib [n]
  (if *enable-trace*
    ; then tracing
    (binding [*parent-call-node*
      (trace-begin-function "example", "fib")]
      (trace-param-expr n)
      (handle-return trace-end-function
        (try
          (trace-expr
            (cond
              (< n 0) 0,
              (< n 2) 1,
              :else (+ (fib (- n 1)) (fib (- n 2))))))
          (catch Throwable t t))))
    ; else no tracing
    (cond
      (< n 0) 0,
      (< n 2) 1,
      :else (+ (fib (- n 1)) (fib (- n 2)))))))
(trace-expr (+ (fib (- n 1)) (fib (- n 2))))

(binding [*parent-call-node*
  (trace-begin-expr '(+ (fib (- n 1)) (fib (- n 2)))]
  (handle-return trace-end-expr
   (try
    (+ (trace-expr (fib (- n 1))) (trace-expr (fib (- n 2))))
    (catch Throwable t t)))

Tracing Features

- tracing activated via `*enable-trace* ⇒ with-trace` macro
- expansion of unknown macros – special treatment for known macros (`clojure.core`)

(with-trace (fib 10)):
given a naïve prime number computation:

```
(defn divisible? [n, k] (= 0 (mod n k)))

(defn divisors [n]
    (doall (filter #(divisible? n %) (range 2 n))))

(defn prime? [n] (empty? (divisors n)))

(defn primes [n]
    (doall (take n (filter prime? (iterate inc 2)))))
```

- timing tree for `(primes 100)`: 
  - `primes` → calls = 1, avg(time) = 321.4561ms
  - `prime?` → calls = 540, avg(time) = 0.5335ms
  - `divisors` → calls = 540, avg(time) = 0.5245ms
  - `divisible?` → calls = 145530, avg(time) = 0.0003ms
Timing Functions (2)

Timing Tree
- nodes for **direct** and **indirect calls**
- only one child node for a function per parent node

```
  f
  /|
 /  |
g  h  g
 h  g
 h  g
```

Timing Node Content
- function information
- invocation count
- minimum, maximum and average time consumption
- total time consumption during tracing
(defn primes [n]
  (if *enable-timing*
    ; then timing
    (let [fn-node
      (get-node-for-call "example", "primes", "cnt", 1)]
      (binding [*parent-node* fn-node]
        (let [start-time (System/nanoTime),
               return-value
               (try
                 (doall
                  (take n (filter prime? (iterate inc 2))))
                 (catch Throwable t t)),
               duration (- (System/nanoTime) start-time)]
          (update-duration-statistic fn-node duration)
          (if (instance? Throwable return-value)
            (throw return-value)
            return-value))))
    ; else no timing
    (doall (take n (filter prime? (iterate inc 2))))))

Development Tools → Timing Functions
Timing Features

- timing activated via `*enable-timing*` ⇒ `with-timing` macro
- summary output on REPL
- Swing GUI for timing tree – `(with-timing (primes 100))`
3 Experiment Tools

- Configuration
- Execution
Parameter Structure Definition
Experiment Definition Language

Concrete Setup
Experiment Setup Language

Result Analysis via Incanter

Execution

(config heuristic
(param alpha :value :float)
(param eval-fn :eval-fn :function [x]))

(heuristic "Greedy"
(alpha* (range 1.0 2.0 0.1))
(eval-fn example/greedy))
Experiment Configuration (2)

- **Experiment definition:**
  - definition of parameters needed for the selected algorithm to run
  - tree of configuration maps $\Rightarrow$ substructuring for different components of the algorithm
  - fixed root configuration map but arbitrary children possible

- **Experiment setup:**
  - setup of concrete experiments with parameter values to be used
  - multiple values per parameter
  - experiments derived via cartesian product or sampling of the given parameter values
(CONFIG <name> <id> <param-list>)

configuration definition:

- <name> name of the configuration macro
- <id> internal id
- <param-list> list of parameter definitions

(CONFIG HEURISTIC :heuristic
  (PARAM ALPHA :alpha :value :float)
  (PARAM EVAL-FN :eval-fn :function [x]))
(PARAM <name> <id> <type> <type-spec>)

parameter definition:

- <name> name of the parameter macro
- <id> internal id
- <type> specification of parameter value type:
  - :value for strings, floats, integers, booleans or keywords
  - <type-spec> $\in\{:\text{string},:\text{float},:\text{integer},:\text{boolean},:\text{keyword}\}$
  - :function for functions – function signature in <type-spec>
  - :config for configuration maps – configuration type id in <type-spec>
  - <type-spec> optional – no checks if not given
(CONFIG HEURISTIC :heuristic
  (PARAM ALPHA :alpha :value :float)
  (PARAM EVAL-FN :eval-fn :function [x]))

(CONFIG ALGORITHM [:algorithm :experiment-parameters]
  (PARAM MAX-ITERATIONS :max-iterations :value :integer)
  (PARAM USED-HEURISTIC :used-heuristic :config :heuristic))

pre-defined root configuration:

(CONFIG EXPERIMENT-SETUP :experiment-setup
  (PARAM REPETITION-COUNT :repetition-count :value :integer)
  (PARAM PARAMETERS :parameters :config :experiment-parameters)
  <other-parameters>
generated from Experiment Definition

for each configuration definition (CONFIG HEURISTIC ...)
  - definition word HEURISTIC: definition of an instance
  - inheritance word HEURISTIC<--: derive configuration from another instance

for each parameter definition (PARAM ALPHA ...)
  - parameter enumeration (ALPHA 1.0, 2.0, 3.0)
  - parameter list (ALPHA* (range 1.0 2.0 0.1))
  - suffix -?? to mark parameter for sampling ALPHA-???, ALPHA-??*
(HEURISTIC "Std"
  (ALPHA 1.0 2.0 3.0)
  (EVAL-FN clojure.core/identity))

(HEURISTIC "Greedy"
  (ALPHA* (range 1.0 4.0 1.0))
  (EVAL-FN example/greedy))

(ALGORITHM "Std+Greedy"
  (MAX-ITERATIONS 100)
  (USED-HEURISTIC "Std", "Greedy"))

(EXPERIMENT-SETUP
  (REPETITION-COUNT 1)
  (PARAMETERS "Std+Greedy"))

⇒ Setup defines 6 experiments
Experiment Execution

Experiment Name

Configuration Path

Experiment Setup

Instance

Experiment Repetition Count

Instance Execution

Instance Run

load

cartesian product or sampling
usage requires implementation of execution function:

```
(defn execute-my-algorithm [instance, parameters]
    <prepare-and-run-algorithm>)
```

- `instance` ... description/id of a problem instance, e.g. file name
- `parameters` ... selected parameters from the experiment setup

experiment execution via

```
(execute-experiment "MySetup", execute-my-algorithm)
```

features:

- progress report with remaining runtime estimation
- parallel execution (same machine)
- automatic storage of result data (filename specification in setup)
Result Analysis with Clojure and Incanter

http://github.com/liebke/incanter
Algorithm implementation support:

- general function definition interception (external configuration)
- interception of: `defn`, `defn-`, `deftype`, `defrecord`

Experimental evaluation support:

- setup specification via DSL
- execution library
library with function definition interception, timing and tracing:

http://github.com/guv/clj-debug

experiment library needs some more polishing:

http://github.com/guv/
Thank you for your attention.