DE GRUYTER
OPEN

JACSM 2016, Vol. 8, No. 2, pp. 115 - 136
10.1515/jacsm-2016-0008

LOW-COST DYNAMIC CONSTRAINT CHECKING FOR THE JVM

Konrad Grzanek

IT Institute, University of Social Sciences
9 Sienkiewicza St., 90-113 Lodz, Poland
kgrzanek@spoleczna.pl

Abstract

Using formal methods for software verification slowly becomes a standard in the industry. Overall it is a good idea to integrate as many checks as possible with the programming language. This is a major cause of the apparent success of strong typing in software, either performed on the compile time or dynamically, on run-time. Unfortunately, only some of the properties of software may be expressed in the type system of event the most sophisticated programming languages. Many of them must be performed dynamically. This paper presents a flexible library for the dynamically typed, functional programming language running in the JVM environment. This library offers its users a close to zero run-time overhead and strong mathematical background in category theory.

Keywords: Formal software verification, software quality, dynamic type-checking, functional programming, category theory, Clojure

1 Introduction

Despite an apparent progress in programming languages theory and practice, the IT industry still experiences problems achieving a desired level of software quality and reliability. M. Thomas [2] mentions that there are from 4 up to 50 bugs (on average) in every 1 thousand lines of production code. This is why the computers are still problematic to rely on in the (not only) safety and mission critical areas of life [1], and the non-critical software is usually hard to use, has got a lowered level of security due to hidden bugs that may even not exhibit themselves on a regular usage basis, but may be exploited by the ones who search for vulnerabilities with an intention to steal information or to introduce other kinds of costly confusion.

This paper is a result of some real-life, production-related experiences and following considerations regarding when and how to perform constraints checks and other kinds

of formal software verification in a dynamically-typed programming language for the *Java Virtual Machine* (*JVM*) environment. We argue that it is a reasonable decision to imply lots of these checks on the time of the program's execution. We also provide a library called *ch* that allows to define and perform some run-time constraint checks. This article may be treated as a good introduction to how this library is implemented and how it can (and should) be used.

1.1 Reasoning About Software Correctness

Scale of contemporary software systems together with the fact that it is intended to run in a multi-tasking environment makes the formal verification methods a strong requirement not an option now. Growing popularity of tools like TLA+ (L. Lamport [5]), temporal logic, and even the hand proofs in software creation process confirm the growing need to become more and more dependent on the beauty of mathematical verification of software systems.

It is a commonly accepted truth that type systems in programming languages are a very strong point in the formal verification of software. Advances in type theory and practice [3, 4] have led to development of programming languages that are particularly effective in catching a variety of common bugs. Let us mention Haskell [9] or Ada here. Lamport [6] says:

„Types have become ubiquitous in computer science. [...] Types do more good than harm in a programming language: they let the compiler catch errors that would otherwise be found only after hours of debugging."

Strong typing means that the expressions in a language contain no implicit data conversions (coercions) that could lead to an unintentional loss of information. *Static type system* is the one in which the compiler (more generally speaking - a static verification tool, the one that reads and analyzes source code) is responsible for making a proof (or dis-proof) that a computer program does not violate any of the type-related invariants that are amenable to pre-run analysis. On the other side, the *dynamically typed languages* are these that leave at least some of the verifications of invariants for the run-time. We should also be aware that there are type-based invariants that may be verified neither statically, nor dynamically. For example, it is impossible to prove the correctness of a famous quick-sort algorithm using a type checker of any kind (for a discussion see [22]). In such cases using a formal method like the TLA+ ahead of the implementation phase alone to prove correctness, or even providing a hand proof is priceless. In any other scenario relying on the type system is a good idea. Typing the specification languages is a completely different problem - for more on that subject, please see [6].

1.2 Discussion on Static and Dynamic Type-Checking

Using a programming language with a static type checker seems a robust method of eliminating bugs in software. When considering only the type-related aspects of a language this statement leads to an immediate conclusion that the dynamic type checking should be avoided at all cost. But the language is much more that that, and there are

multiple reasons why the dynamically typed programming languages have made such a great success in the industry. Among the others, the dynamic languages:

- Allow to perform immediate tests of the functionality being implemented by using *REPL* (*Read-Eval-Print Loop*).
- In the case of languages from the Lisp family, like Clojure [7, 8] it is even possible to compile new functionality without stopping the running program. The REPL execution takes part in the same environment in which the production software runs, we have no debug-release cycle here, and that feature alone is a great productivity booster.
- A lack of static type system allows to define programming constructs that are intended to be run in contexts that would be very hard or even impossible to describe using a set of static type-related constraints. Some Lisp *macros* are good example here.

Even when we decide to rely on a static type checker we should be aware that the extent to which we will be able to verify programs using this tool is limited to what can be expressed in the rules of the type system, and this extent has bounds. Other checks/verifications/proofs must be performed anyway either on the run-time or by hand proving. Static type systems generally verify that the elements of a system fit together as a structure, and only sometimes can prove or disprove their homeostasis and well-functioning over a period of time.

1.3 Functional Programming

Another means of establishing high level of software quality and reliability is using functional programming languages like Haskell, Clojure, Erlang. The impact of the immutability of data structures on software correctness, its predictability and a relative ease of searching for bugs may be at least as big as using the type checker to find the mistakes statically. Out of the mentioned three languages two are dynamically typed (Erlang, Clojure) and they are highly successful even in mission critical domains.

1.4 Existing Solutions for Clojure

Clojure programming language [7, 8] is one of the most interesting contemporary JVM-targeted languages. It belongs to Lisp family, and - as its elders - is dynamically typed. This section presents current approaches that exist in this language, and that are related to the problems formulated above.

One attempt to employ some kind of static type checking is *clojure/core.typed* library [20]. This solution uses type annotations and a static type checker. Unfortunately, the realization has severe performance problems as described in the discussion [21]. Moreover, due to the reasons described in the paragraphs above a full static type checker does not make a perfect fit with respect to the pragmatics of using Lisps and to the overall idea of implementing software quickly.

Another similar library that suffers similar problems is *prismatic/schema* [19]. In the case of this solution, we find it better in terms of the overall usability and performance, but the type annotations a kind of get in the way with the normal ways of using Lisp, that is - writing as much as possible using *s-expressions*.

Finally, the most promising project in this domain is Cognitect's *clojure.spec* [17, 18]. It may become a de-facto standard specification tool, but its goals are slightly different than the ones we are looking at. Its development process is in its early stages, as the adoption in the industry.

These considerations led us to the idea of creating a custom solution - Clojure library meeting the requirements of dynamic specification/type/invariants validator, with the following assumptions:

- being deeply rooted in functional programming [9] and using notions from the category theory [10],
- consistency with the Lisp nature of Clojure programming language, by using s-expressions only (Lisp as a „big ball of mud"),
- relying on fast dynamic type-checking routines of the JVM - the *instanceof* operator,

Our solution is called *kongra/ch* [12, 13], shortly *ch*, and it has been successfully used in *kongra/prelude* [14, 15] and *aptell* [16] projects. The following sections of the paper are detailed description of its implementation and possible use.

2 Essentials of the ch Library

Every predicate check uses the following procedure to generate a message describing the value, together with its type, that violates the check:

```
(defn chmsg
  [x]
  (with-out-str
    (print "Illegal value ")    (pr x)
    (print " of type ") (pr (class x))))
```

When executed in the REPL the procedure works as follows:

```
user> (chmsg 123)
"Illegal value 123 of type java.lang.Long"
```

or, for nil values:

```
user> (chmsg nil)
"Illegal value nil of type nil"
```

The most essential syntactic structure in the *ch* library is a $(ch\ldots)$ form. Due to the implementation issues the form is intended to be used both as an assertion and as a boolean-valued function. The definition begins with a supporting function *pred-call-form*:

```
(defn- pred-call-form
  ([form x]
   (let [form (if (symbol? form) (vector form) form)]
     (seq (conj (vec form) x))))

  ([form _ x]
   (let [form (if (symbol? form) (vector form) form)]
     (concat form (list nil x)))))
```

and the actual *ch* macro that uses the function to generate a target s-expression, either an assertion or a predicate-like call:

```
(defmacro ch {:style/indent 1}
  ([pred x]
   (let [x'    (gensym "x__")
         form (pred-call-form pred x')]
     `(let [~x' ~x] (assert ~form (chmsg ~x')) ~x')))

  ([pred #_ be-pred _ x]
   (let [form (pred-call-form pred x)]
     `(boolean ~form))))
```

To see, what actually happens when a compiler encounters the (*ch*...) form, please take a look at the expression *(ch nil? 123)*. This expression evaluates the function *nil?* belonging to the Clojure standard library against the argument, integral (Long) value 123. The target form is

```
(let [x__11622 123]
  (assert (nil? x__11622) (chmsg x__11622))
  x__11622)
```

The compiler introduces an additional local variable x__11622 that holds the value of an evaluated 123 input value (expression from compiler's point of view) and executes the (*assert*...) on it using the passed *nil?* function an assert's predicate. After a successful evaluation the evaluated value of x__11622 is returned resulting in the desired behavior. This time it's failure, because 123 is not a nil value:

```
user> (ch nil? 123)
AssertionError Assert failed: Illegal value 123 of type
    java.lang.Long
(nil? x__10994)  kongra.ch/eval10995
    (form-init3881948826253525319.clj:17)
```

If we decide to use (*ch*...) using its predicate "nature", like *(ch nil? :as-pred 123)*, we get:

```
(boolean (nil? 123))
```

and the evaluation of the form ends with *false* value being returned.

Now, let's talk about the performance. All the following performance benchmarks were taken in the commodity hardware environment: Intel i7-5500U, 16 GB of RAM, Ubuntu 14.04 64-bit using the awesome *criterium*[1] library for benchmarking Clojure codes. Additionally we have: CIDER 0.14.0 (Berlin), nREPL 0.2.12, Clojure 1.8.0, Java 1.8.0_121. At first a simple expression:

```
user> (quick-bench (nil? 123))
Evaluation count : 52064214 in 6 samples of 8677369 calls.
             Execution time mean : 1,113992 ns
    Execution time std-deviation : 0,041270 ns
   Execution time lower quantile : 1,058363 ns ( 2,5%)
```

[1] https://github.com/hugoduncan/criterium

```
   Execution time upper quantile : 1,160112 ns (97,5%)
                  Overhead used : 10,475943 ns
```

and the corresponding predicate form of (*ch*...):

```
user> (quick-bench (ch nil? :as-pred 123))
Evaluation count : 52235052 in 6 samples of 8705842 calls.
             Execution time mean : 1,078551 ns
    Execution time std-deviation : 0,115450 ns
   Execution time lower quantile : 0,942867 ns ( 2,5%)
   Execution time upper quantile : 1,215710 ns (97,5%)
                  Overhead used : 10,475943 ns
```

Also for the assertion:

```
user> (quick-bench (ch nil? nil))
Evaluation count : 52466976 in 6 samples of 8744496 calls.
             Execution time mean : 1,081879 ns
    Execution time std-deviation : 0,135884 ns
   Execution time lower quantile : 0,977794 ns ( 2,5%)
   Execution time upper quantile : 1,262656 ns (97,5%)
                  Overhead used : 10,475943 ns
```

We can clearly see that there is no apparent overhead of the call. This is only an introductory example, so it is impossible to reason now about the target performance loss when applying this approach to a production software. This will be discussed in further parts of the paper.

3 Generator of Ch(eck)s

The library would be far from being useful, if the user would be forced to use raw (*ch*...) forms everywhere. Instead, we introduce the *(defch . . .)* macro that allows the programmer to define his own named checks. The macro code goes as follows:

```
(defmacro defch {:style/indent 1}
  ([chname form]
   (let [x     (gensym "x__")
         form+ (append-arg form x) ]
     `(defch ~chname [~x] ~form+)))

  ([chname args form]
   (assert (vector? args))
   (let [args+ (insert-noparam args)
         form+ (insert-noarg   form)]
     `(defmacro ~chname {:style/indent 1}
        (~args  ~form)
        (~args+ ~form+)))))
```

To do its job, the macro performs some manipulations with the arguments and the shape of the target form, which is a subsequent macro in this case. So we may say that *defch* is a macro-writing macro.

The arguments must be enhanced do support a non-assertion (predicate) use, because - as in the case of raw *ch* - we tend to operate both in assertion and in predicate mode. The arguments of the target macro are prepared using the following *insert-noparam*:

```
(defn- insert-noparam
  [params]
  (vec (concat (butlast params)
               (list '_)
               (when (seq params) (list (last params))))))
```

while the two following procedures prepare the predicate target form:

```
(defn- insert-noarg
  [form]
  (let [;; lein eastwood passes a wrapper (sequence <form>),
        ;; let's strip it down:
        form (if (= (first form) 'sequence) (second form) form)
        [ccseq [cconcat & cclists]] form]
    (assert (=  ccseq        'seq) (str "Illegal ccseq "
       ccseq " in " form))
    (assert (=  cconcat    'concat) (str "Illegal cconcat "
       cconcat " in " form))
    (assert (>= (count cclists) 2) (str "Illegal cclists "
       cclists " in " form))
    (let [lsts   (butlast  cclists)
          lst    (last     cclists)
          noarg '(list        'nil)]
      '(seq (concat ~@lsts ~noarg ~lst)))))

(defn- append-arg
  [form x]
  (let [;; lein eastwood passes a wrapper (sequence <form>),
        ;; let's strip it down:
        form (if (= (first form) 'sequence) (second form) form)
        [ccseq [cconcat & cclists]] form]
    (assert (=  ccseq        'seq) (str "Illegal ccseq "
       ccseq " in " form))
    (assert (=  cconcat    'concat) (str "Illegal cconcat "
       cconcat " in " form))
    (assert (>= (count cclists) 2) (str "Illegal cclists "
       cclists " in " form))
    (let [arg '(list ~x)]
      '(seq (concat ~@cclists ~arg)))))
```

Finally we may take a look at how this entirety works together.

4 Unit Type Ch(eck)

Unit type is a very useful type in many programming languages. Unit type has exactly one value. It is so called *terminal object* in category of types and typed functions. In

some programming languages (e.g. Haskell) the unit type is expressed as (), while in others (like C/C++/Java) a *void* keyword is used to express something related, namely the fact that the procedure does not return any value. The latter approach may be somewhat informally referred to as a means to express a lack of information at the output of a procedure, and this follows the original nature of the unit type in category theory - a type with only one object carries on no information.

In Clojure, as in Java, we traditionally use nil as a representation of unit type. The check for nil-ness is defined as follows:

```
;; UNIT (NIL)
(defch chUnit [x] `(ch nil? ~x))
```

This form introduces a macro named *chUnit* that my be used in the following two ways:

```
user> (chUnit 1)
AssertionError Assert failed: Illegal value 1 of type
    java.lang.Long
(clojure.core/nil? x__11646)  kongra.ch/eval11647
    (form-init3881948826253525319.clj:83)
```

or

```
user> (chUnit :as-pred 1)
false
```

Additionally we introduce complementary ch(eck)s for non-nil values:

```
;; NON-UNIT (NOT-NIL)
(defn not-nil? {:inline (fn [x] `(if (nil? ~x) false true))}
                           [x]  (if (nil?  x) false true))
(defch chSome [x] `(ch not-nil? ~x))
```

Their cost is as abysmal as for nil? check, as presented in one of the previous sections. With the following definitions of test procedures:

```
(defn foo [x] (chUnit x))
(defn goo [x] (chSome x))
```

we have:

```
user> (quick-bench (foo nil))
Evaluation count : 35042178 in 6 samples of 5840363 calls.
             Execution time mean : 6,895234 ns
    Execution time std-deviation : 0,280141 ns
   Execution time lower quantile : 6,617001 ns ( 2,5%)
   Execution time upper quantile : 7,299458 ns (97,5%)
                   Overhead used : 10,475943 ns
```

and:

```
user> (quick-bench (goo 123))
Evaluation count : 34250304 in 6 samples of 5708384 calls.
             Execution time mean : 7,283686 ns
    Execution time std-deviation : 0,130028 ns
   Execution time lower quantile : 7,138009 ns ( 2,5%)
```

```
Execution time upper quantile : 7,444272 ns (97,5%)
                Overhead used : 10,475943 ns
```

To look more deeply in what happens under the hood in these test procedures, let's use *no.disassemble*[2] library to view the resulting byte-code for *foo*:

```
// Method descriptor #11 (Ljava/lang/Object;)Ljava/lang/Object;
 // Stack: 8, Locals: 2
 public static java.lang.Object invokeStatic(java.lang.Object
     x);
    0  aload_0 [x]
    1  aconst_null
    2  astore_0 [x]
    3  astore_1 [x__16079]
    4  aload_1 [x__16079]
    5  aconst_null
    6  invokestatic
        clojure.lang.Util.identical(java.lang.Object,
        java.lang.Object) : boolean [17]
    9  ifeq 18
   12  aconst_null
   13  pop
   14  goto 79
   17  pop
   18  new java.lang.AssertionError [19]
   21  dup
   22  getstatic kongra.ch$foo.const__1 : clojure.lang.Var [23]
   25  invokevirtual clojure.lang.Var.getRawRoot() :
       java.lang.Object [29]
   28  checkcast clojure.lang.IFn [31]
   31  ldc <String "Assert failed: "> [33]
   33  getstatic kongra.ch$foo.const__2 : clojure.lang.Var [36]
   36  invokevirtual clojure.lang.Var.getRawRoot() :
       java.lang.Object [29]
   39  checkcast clojure.lang.IFn [31]
   42  aload_1 [x__16079]
   43  invokeinterface
       clojure.lang.IFn.invoke(java.lang.Object) :
       java.lang.Object [39] [nargs: 2]
   48  ldc <String "\n"> [41]
   50  getstatic kongra.ch$foo.const__3 : clojure.lang.Var [44]
   53  invokevirtual clojure.lang.Var.getRawRoot() :
       java.lang.Object [29]
   56  checkcast clojure.lang.IFn [31]
   59  getstatic kongra.ch$foo.const__4 : java.lang.Object [48]
   62  invokeinterface
       clojure.lang.IFn.invoke(java.lang.Object) :
       java.lang.Object [39] [nargs: 2]
```

[2]https://github.com/gtrak/no.disassemble

```
67  invokeinterface
    clojure.lang.IFn.invoke(java.lang.Object,
    java.lang.Object, java.lang.Object, java.lang.Object) :
    java.lang.Object [51] [nargs: 5]
72  invokespecial
    java.lang.AssertionError(java.lang.Object) [54]
75  checkcast java.lang.Throwable [56]
78  athrow
79  aload_1 [x__16079]
80  aconst_null
81  astore_1
82  areturn
```

When Clojure direct linking is enabled, we have an even more optimized code like:

```
public static java.lang.Object invokeStatic(java.lang.Object
    x);
 0  aload_0 [x]
 1  aconst_null
 2  astore_0 [x]
 3  astore_1 [x__12541]
 4  aload_1 [x__12541]
 5  aconst_null
 6  invokestatic
     clojure.lang.Util.identical(java.lang.Object,
     java.lang.Object) : boolean [17]
 9  ifeq 19
12  getstatic java.lang.Boolean.FALSE : java.lang.Boolean
    [23]
15  goto 22
18  pop
19  getstatic java.lang.Boolean.TRUE : java.lang.Boolean
    [26]
22  dup
23  ifnull 37
26  getstatic java.lang.Boolean.FALSE : java.lang.Boolean
    [23]
29  if_acmpeq 38
32  aconst_null
33  pop
34  goto 92
37  pop
38  new java.lang.AssertionError [28]
41  dup
42  ldc <String "Assert failed: "> [30]
44  iconst_3
45  anewarray java.lang.Object [32]
48  dup
49  iconst_0
50  aload_1 [x__12541]
51  invokestatic
```

```
    kongra.ch$chmsg.invokeStatic(java.lang.Object) :
    java.lang.Object [36]
54  aastore
55  dup
56  iconst_1
57  ldc <String "\n"> [38]
59  aastore
60  dup
61  iconst_2
62  iconst_1
63  anewarray java.lang.Object [32]
66  dup
67  iconst_0
68  getstatic kongra.ch$foo.const__5 : java.lang.Object [42]
71  aastore
72  invokestatic
    clojure.lang.ArraySeq.create(java.lang.Object[]) :
    clojure.lang.ArraySeq [48]
75  invokestatic
    clojure.core$pr_str.invokeStatic(clojure.lang.ISeq) :
    java.lang.Object [53]
78  aastore
79  invokestatic
    clojure.lang.ArraySeq.create(java.lang.Object[]) :
    clojure.lang.ArraySeq [48]
82  invokestatic
    clojure.core$str.invokeStatic(java.lang.Object,
    clojure.lang.ISeq) : java.lang.Object [58]
85  invokespecial
    java.lang.AssertionError(java.lang.Object) [61]
88  checkcast java.lang.Throwable [63]
91  athrow
92  aload_1 [x__12541]
93  aconst_null
94  astore_1
95  areturn
```

The code is actually a call to the *(assert ...)* form as defined in the original *(ch ...)* mechanism. With this in mind we may be certain to get a very efficient check-instrumenting code in all the places we use the *ch* library, and the actual cost of every check depends solely on the nature (and cost) of the predicates he uses. It is the responsibility of the programmer to keep them as cheap as possible. From what is well known the *instanceof* predicates in the JVM are particularly fast and cheap both on the CPU and the memory side. The following section discusses a way the *ch* library approaches the type-checks.

5 Class Membership Ch(eck)s

To come up with a proper, fast, run-time type checking, we first provide the following macro that makes the *instance?* call:

```
;; CLASS MEMBERSHIP
(defch chC [c x] `(ch (instance? ~c) ~x))
```

Then we are ready to define another utility macro-writing macro *defchC* that allows the programmer to define his own type checks (besides the ones defined in the *ch* library, see Appendix A):

```
(defmacro defchC
  [chname c]
  (let [x (gensym "x__")]
    `(defch ~chname [~x] `(chC ~~c ~~x))))
```

Among the others the check for *java.lang.Long* type is defined like: *(defchC chLong Long)*. Now we can turn the procedure *(defn foo [x] x)* into *(defn foo [x] (chLong x))*. The resulting procedure has the performance profile as in the following benchmarking result:

```
user> (quick-bench (foo 1))
Evaluation count : 198219084 in 6 samples of 33036514 calls.
             Execution time mean : 1,033696 ns
    Execution time std-deviation : 0,010647 ns
   Execution time lower quantile : 1,012954 ns ( 2,5%)
   Execution time upper quantile : 1,040669 ns (97,5%)
                   Overhead used : 2,032505 ns
```

Similarly for *java.lang.String* we have:

```
user> (quick-bench (chString ""))
Evaluation count : 88947072 in 6 samples of 14824512 calls.
             Execution time mean : 4,806284 ns
    Execution time std-deviation : 0,094706 ns
   Execution time lower quantile : 4,724241 ns ( 2,5%)
   Execution time upper quantile : 4,942071 ns (97,5%)
                   Overhead used : 2,032505 ns
```

These benchmarks shows two things:

- The modern JVM has perfect ways to optimize type checks up to a level that is almost hard to notice from a point of a programmer who writes common code, even in the performance-critical parts.
- Our approach makes no overhead when calling these mechanisms and making the JVM actually do its job.

6 Object Type Equality Ch(eck)s

In a common programmers' practice we often have to ensure two objects have exactly the same type. Check *chLike*, as defined below, serves exactly that:

```
;; OBJECT TYPE EQUALITY
(defmacro chLike* [y x] `(identical? (class ~y) (class ~x)))
(defch    chLike  [y x] `(ch (chLike* ~y) ~x))
```

Let's take look at its performance benchmark, here for two Strings:

```
user> (quick-bench (chLike "aaa" "bbb"))
Evaluation count : 88657260 in 6 samples of 14776210 calls.
             Execution time mean : 4,793546 ns
    Execution time std-deviation : 0,079737 ns
   Execution time lower quantile : 4,724969 ns ( 2,5%)
   Execution time upper quantile : 4,927724 ns (97,5%)
                   Overhead used : 2,032505 ns
```

7 Product (Pair/Tuple) and Co-Product (Discriminated Union Type) Ch(eck)s

To apply more compound checks that use logical operators we define the following generator of predicate checks:

```
(defmacro ch*
  [op chs x]
  (assert (vector? chs) "Must be a chs vector in (ch| ...)")
  (assert (seq     chs) "(ch| ...) must contain some chs"  )
  `(~op ~@(map #(pred-call-form % nil x) chs)))
```

The generator is used to define checks for *tuple* types and for *discriminated union* types:

```
;; PRODUCT (TUPLE)
(defch ch& [chs x] `(ch (ch* and ~chs) ~x))

;; CO-PRODUCT (DISCRIMINATED UNION TYPE)
(defch ch| [chs x] `(ch (ch* or  ~chs) ~x))
```

A special case here is a co-product of exactly two types, known as *Either a b* type constructor in some languages (e.g. Haskell), and its variant - the *Maybe a* type constructor, that can be defined as $type Maybe\, a = Either\, ()\, a$. The *ch* library specifies them as follows:

```
(defch chEither [chl chr x] `(ch| [~chl  ~chr]    ~x))
(defch chMaybe  [ch      x] `(chEither chUnit ~ch ~x))
```

And an example benchmark:

```
user> (quick-bench (chEither chString chLong 1))
Evaluation count : 91910928 in 6 samples of 15318488 calls.
             Execution time mean : 4,879827 ns
    Execution time std-deviation : 0,660203 ns
   Execution time lower quantile : 4,433577 ns ( 2,5%)
   Execution time upper quantile : 5,924470 ns (97,5%)
                   Overhead used : 2,032505 ns
```

Again, we cannot see any significant impact on the performance, other than few nanoseconds.

8 Registry of Ch(eck)s

Additionally the *ch* library provides a registry of checks, that helps the programmer to understand, what kind of checks an object or a collection of objects fulfill. The registry is actually a mapping from string (a name) into a check:

```
(def ^:private CHS (atom {}))
```

To register a new check we use the following macro:

```
(defmacro regch
  [ch]
  (assert (symbol? ch))
  (let [x (gensym "x__")]
    `(regch* ~(str ch) (fn [~x] ~(pred-call-form ch nil x)))))
```

together with its back-end:

```
(defn regch*
  [chname ch]
  (chUnit
   (do
     (assert (string? chname))
     (assert (fn?         ch))
     (swap! CHS
            (fn [m]
              (when (m chname)
                (println "WARNING: chname already in use:"
                    chname))
              (assoc m chname ch))) nil)))
```

Now we may use *chs* function:

```
(defn chs
  ([]
   (chSet (apply sorted-set (sort (keys @CHS)))))

  ([x]
   (chSet (->> @CHS
               (filter (fn [[_ pred]] (pred x)))
               (map first)
               (apply sorted-set))))
  ([x & xs]
    (chSet (->> (cons x xs) (map chs) (apply
        cset/intersection)))))
```

to reach for the information, e.g.:

```
user> (chs 1)
#{"chInteger" "chLong" "chNumber" "chRational"}
```

```
user> (chs "a")
#{"chString"}
user> (chs [1 2 3])
#{"chAssoc" "chColl" "chCounted" "chIfn" "chIndexed"
    "chJavaColl" "chJavaList" "chLookup" "chReversible"
    "chSeqable" "chSequential" "chVec"}
user> (chs inc)
#{"chFn" "chIfn"}
```

There is also a possibility to ask for checks common for a set of objects:

```
user> (chs 1)
#{"chInteger" "chLong" "chNumber" "chRational"}
user> (chs 1.23)
#{"chDouble" "chFloat" "chNumber"}
user> (chs 3/4)
#{"chNumber" "chRatio" "chRational"}
user> (chs 1 1.23 3/4)
#{"chNumber"}
```

Using the information provided the programmer can make a decision on what checks to use in a particular situation.

9 Example Use in Production Setting

The *ch* library is used extensively in production. Among the others it was used to tag some of the elements of *kongra/prelude* package. With the following namespace declaration:

```
(ns kongra.prelude.search
  (:require [kongra.ch      :refer :all]
            [kongra.prelude :refer :all]))
```

we define tree-search routines in the *kongra.prelude.search* namespace. In the first place we define a combiner function that controls the tree search process, by performing order-wise concatenation of search space elements. The concatenation operates on sequences and returns a sequence, thus the use of *chSeq* in the following code:

```
;; COMBINERS
(deftype Comb [f]
  clojure.lang.IFn
  (invoke [_ nodes new-nodes]
    (chSeq (f (chSeq nodes) (chSeq new-nodes)))))
```

For the combiner we define a *chComb* ch(eck) and a proper constructor (*consComb*):

```
(defchC chComb Comb)
(defn consComb [f] (Comb. (chIfn f)))
```

The combiners for breath-first and depth-first search strategies are defined as follows:

```
(def breadth-first-combiner      (consComb    concat))
(def lazy-breadth-first-combiner (consComb lazy-cat'))
```

```
(def depth-first-combiner         (consComb #(concat   %2 %1)))
(def lazy-depth-first-combiner    (consComb #(lazy-cat %2 %1)))
```

Using exactly the same pattern we define an abstraction for goal functions

```
;; GOAL
(deftype Goal [f]
  clojure.lang.IFn
  (invoke [_ x] (boolean (f x))))

(defchC chGoal 'Goal)
(defn consGoal [f] (Goal. (chIfn f)))
```

and for adjacency generators for the tree structure:

```
;; ADJACENCY
(deftype Adjs [f]
  clojure.lang.IFn
  (invoke [_ x] (chSeq (f x))))

(defchC chAdjs Adjs)
(defn consAdjs [f] (Adjs. (chIfn f)))
```

Finally the general *tree-search* procedure uses all the checks defined earlier as presented in the following listing:

```
(defn tree-search
  [start goal? adjs comb]
  (chGoal goal?) (chAdjs adjs) (chComb comb)
  (chMaybe chSome
    (loop [nodes (list start)]
      (when (seq nodes)
        (let [obj (first nodes)]
          (if (goal? obj)
            obj
            (recur (comb (rest nodes) (adjs obj)))))))))
```

Both major search strategies have the implementations like:

```
(defn breadth-first-search
  [start goal? adjs]
  (chMaybe chSome
    (tree-search start
                 (chGoal          goal?)
                 (chAdjs           adjs)
                breadth-first-combiner)))
```

and:

```
(defn depth-first-search
  [start goal? adjs]
  (chMaybe chSome
    (tree-search start
```

```
                (chGoal        goal?)
                (chAdjs         adjs)
                depth-first-combiner)))
```

A very useful procedure *breadth-first-tree-levels* that returns consecutive depth levels of a tree also uses the mechanisms.

```
(defn breadth-first-tree-levels
  [start adjs]
  (chAdjs adjs)
  (chSeq (->> (list               start)
              (iterate #(mapcat adjs %))
              (map              chSeq')
              (take-while          seq))))
```

And finally the following traversal mechanism returns a lazily evaluated sequence of all tree nodes visited according to a breadth-first strategy:

```
(defn breadth-first-tree-seq
  ([start adjs]
   (chAdjs adjs)
   (chSeq (apply concat (breadth-first-tree-levels start
       adjs))))

  ([start adjs depth]
   (chAdjs     adjs)
   (chPosLong depth)
   (chSeq (->> (breadth-first-tree-levels start adjs)
               (take  depth)
               (apply concat)))))
```

10 Plans for the Future Development

Performing the dynamic (run-time) checks for various constraints, including the verification of types is not enough to ensure proper integrity of software projects. Many factors come to mind here, including:

- Multiple versions of libraries that software projects depend upon, together with the information on the actual use of these dependencies, circularity of dependencies, and general lack of reliable sources of coherent packages (libraries) bundled together.
- Lack of the ability to effectively model highly complex systems, like the Java 8 Language Specification, being turned on into a working compiler or at least a static analyzer working according to the rules present in the specification. The estimated amount of work for creating Java compiler is hundreds of man-years, while it would be great to be able to perform activities like that in time an order of magnitude shorter. Problems of this kind were already discussed by us in [11].

These issues and the possibility to solve them in an uniform way will be subjects of our further research activities. We tend to make *ch* library a part of the solution.

References

1. Thomas M., 2016, *How Can Software Be So Hard?*, Gresham College Lecture, Feb. 2016, https://www.youtube.com/watch?v=VfRVz1iqgKU
2. Thomas M., 2015, *Should We Trust Computers*, Gresham College Lecture, Oct. 2015, https://www.youtube.com/watch?v=8SZfjvlbpMw
3. Pierce B.C., 2002, *Types and Programming Languages, 1st Edition*, MIT Press, ISBN-10: 0262162091, ISBN-13: 978-0262162098
4. Pierce B.C., 2004, *Advanced Topics in Types and Programming Languages*, MIT Press, ISBN-10: 0262162288, ISBN-13: 978-0262162289
5. Lamport L., 2002, *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*, Addison Wesley, ISBN: 0-321-14306-X
6. Lamport L., Paulson L.L., 1999, *Should Your Specification Language Be Typed?*, ACM Transactions on Programming Languages and Systems, Vol. 21, No. 3, pp. 502-526
7. Emerick Ch., Carper B., Grand Ch., 2012, *Clojure Programming*, O'Reilly Media Inc., ISBN: 978-1-449-39470-7
8. Fogus M., Houser Ch., 2014, *The Joy of Clojure*, Manning Publications; 2 edition, ISBN-10: 1617291412, ISBN-13: 978-1617291418
9. Bird R., Wadler R., 1988, *Introduction to Functional Programming*. Series in Computer Science (Editor: C.A.R. Hoare), Prentice Hall International (UK) Ltd
10. Awodey S., 2010, *Category Theory, Second Edition*, Oxford University Press
11. Grzanek K., 2012, *Prerequisites for Effective Requirements Management*, Journal of Applied Computer Science Methods, Vol. 4, No 1, pp. 21-28
12. Grzanek K., 2017, *ch GitHub Repository*, https://github.com/kongra/ch
13. Grzanek K., 2017, *ch Clojars Page*, https://clojars.org/kongra/ch
14. Grzanek K., 2017, *prelude GitHub Repository*, https://github.com/kongra/prelude
15. Grzanek K., 2017, *prelude Clojars Page*, https://clojars.org/kongra/prelude
16. Grzanek K., 2017, *aptell GitHub Repository*, https://github.com/kongra/aptell
17. Clojure Team, 2017, *clojure.spec Rationale*, https://clojure.org/about/spec
18. Clojure Team, 2017, *clojure.spec Guide*, https://clojure.org/guides/spec
19. Plumatic, 2017, *prismatic/schema GitHub Repository*, https://github.com/plumatic/schema

20. Clojure Team, 2017, *clojure/core.typed GitHub Repository*, https://github.com/clojure/core.typed

21. CircleCI, 2015, *Why we're no longer using core.typed*, https://circleci.com/blog/why-were-no-longer-using-core-typed/

22. Hacker News, 2015, *Why were no longer using core.typed - discussion*, https://news.ycombinator.com/item?id=10271149

A Appendix: Selected Pre-defined Checks

In the beginning we define the following type checks for basic types (classes) belonging to the standard Clojure library and/or forming the Clojure run-time environment:

```
;; COMMON CHS
(defchC chAgent           clojure.lang.Agent)
(defchC chAtom             clojure.lang.Atom)
(defchC chASeq             clojure.lang.ASeq)
(defchC chBoolean                    Boolean)
(defchC chDeref          clojure.lang.IDeref)
(defchC chDouble                      Double)
(defchC chIndexed       clojure.lang.Indexed)
(defchC chLazy          clojure.lang.LazySeq)
(defchC chLong                          Long)
(defchC chLookup        clojure.lang.ILookup)
(defchC chRef               clojure.lang.Ref)
(defchC chSeqable       clojure.lang.Seqable)
(defchC chSequential clojure.lang.Sequential)
```

Another family of the basic checks is the checks defined upon some essential predicates belonging to the standard library of the language:

```
(defch  chAssoc           '(ch associative?))
(defch  chChar                   '(ch char?))
(defch  chClass                 '(ch class?))
(defch  chColl                   '(ch coll?))
(defch  chCounted              '(ch counted?))
(defch  chDecimal              '(ch decimal?))
(defch  chDelay                  '(ch delay?))
(defch  chFloat                  '(ch float?))
(defch  chFn                        '(ch fn?))
(defch  chFuture               '(ch future?))
(defch  chIfn                      '(ch ifn?))
(defch  chInteger              '(ch integer?))
(defch  chKeyword              '(ch keyword?))
(defch  chList                    '(ch list?))
(defch  chMap                      '(ch map?))
(defch  chNumber               '(ch number?))
(defch  chRatio                  '(ch ratio?))
(defch  chRational           '(ch rational?))
```

```
(defch  chRecord               '(ch record?))
(defch  chReduced             '(ch reduced?))
(defch  chReversible       '(ch reversible?))
(defch  chSeq                    '(ch seq?))
(defch  chSorted              '(ch sorted?))
(defch  chString              '(ch string?))
(defch  chSymbol              '(ch symbol?))
(defch  chVar                    '(ch var?))
(defch  chVec                 '(ch vector?))
```

Finally we introduce type checks for few basic Java collection interfaces. It is worth mentioning, that all Clojure collections implement one of these interfaces:

```
(defchC chJavaColl      java.util.Collection)
(defchC chJavaList            java.util.List)
(defchC chJavaMap              java.util.Map)
(defchC chJavaSet              java.util.Set)
```

B Appendix: Selected Tests

The *ch* library is covered with unit tests. Here we present few of them to give the reader another opportunity to get familiar with syntax and behavior of the library. In the following codes we use the namespace definition as below:

```
(ns kongra.ch-test
  (:require [clojure.test :refer :all]
            [kongra.ch :refer :all]))
```

First of all let's define few types with their accompanying type checks:

```
(deftype X []) (defchC chX X)
(deftype Y []) (defchC chY Y)
(deftype Z []) (defchC chZ Z)
```

We also define the following simple checks of various kinds:

```
(defch chMaybeX      '(chMaybe  chX                      ))
(defch chEitherXUnit '(chEither chX chUnit               ))
(defch chEitherXY    '(chEither chX    chY               ))
(defch chXYZ         '(ch| [chX chY chZ]                 ))
(defch chMaybeLike1  '(chMaybe (chLike 1                )))
(defch chEitherLC    '(chEither (chC Long) (chC Character)))
(defch chEitherLC'
  '(chEither (ch (instance? Long)) (ch (instance? Character))))
```

as well as the compound one:

```
(defch chCompound1
  '(chEither
       (chMaybe  (chLike "aaa"))
       (chEither (chMaybe (ch (instance? Long)))
                 (chMaybe (ch (instance? Character))))))
```

Here the test cases follow:

```
(testing "(ch ...)"
    (is (thrown? AssertionError (ch (nil?)       1)))
    (is (nil?                   (ch (nil?)     nil)))
    (is (false?                 (ch (nil?) nil   1)))
    (is (true?                  (ch (nil?) nil nil))))

(testing "(ch ...) with symbolic preds"
    (is (thrown? AssertionError (ch  nil?        1)))
    (is (nil?                   (ch  nil?      nil)))
    (is (false?                 (ch  nil?  nil   1)))
    (is (true?                  (ch  nil?  nil nil))))

(testing "(chC ...)"
    (is (= ""                   (chC String      "")))
    (is (thrown? AssertionError (chC String       1)))
    (is (thrown? AssertionError (chC String     nil)))
    (is (true?                  (chC String nil  "")))
    (is (false?                 (chC String nil   1)))
    (is (false?                 (chC String nil nil))))

(testing "(defchC ...)"
    (is                         (chX      (X.)))
    (is (thrown? AssertionError (chX        1)))
    (is (thrown? AssertionError (chX      nil)))
    (is (true?                  (chX nil (X.))))
    (is (false?                 (chX nil    1)))
    (is (false?                 (chX nil nil))))

(testing "(chLike ...)"
    (is                         (chLike     1          2))
    (is (thrown? AssertionError (chLike     1     "aaa")))
    (is (thrown? AssertionError (chLike "aaa"         2)))
    (is (thrown? AssertionError (chLike     1       nil)))

    (is (true?                  (chLike   2/3 nil   3/4)))
    (is (false?                 (chLike     1 nil "aaa")))
    (is (false?                 (chLike "aaa" nil     2)))
    (is (false?                 (chLike     1 nil   nil))))

(testing "(chUnit ...)"
    (is (nil?                   (chUnit     nil)))
    (is (thrown? AssertionError (chUnit       1)))

    (is (true?                  (chUnit nil nil)))
    (is (false?                 (chUnit nil ""))))

(testing "(chSome ...)"
    (is                         (chSome        1))
```

```
    (is (thrown? AssertionError (chSome      nil)))

    (is (true?                  (chSome  nil  "")))
    (is (false?                 (chSome  nil nil))))

(testing "(chMaybe ...)"
    (is (nil?                   (chMaybe chX      nil)))
    (is                         (chMaybe chX      (X.)))
    (is (thrown? AssertionError (chMaybe chX      (Y.))))
    (is (true?                  (chMaybe chX nil  nil)))
    (is (true?                  (chMaybe chX nil  (X.))))
    (is (false?                 (chMaybe chX nil  (Y.))))

    (is (nil?                   (chMaybe chUnit nil)))
    (is (thrown? AssertionError (chMaybe chUnit (X.))))
    (is (thrown? AssertionError (chMaybe chUnit (Y.)))))

(testing "(chEither ...)"
    (is (nil?                   (chEither chX chUnit  nil)))
    (is                         (chEither chX chUnit  (X.)))
    (is (thrown? AssertionError (chEither chX chUnit  (Y.))))
    (is                         (chEither chX chY     (X.)))
    (is                         (chEither chX chY     (Y.)))
    (is (thrown? AssertionError (chEither chX chY    (Z.))))
    (is (thrown? AssertionError (chEither chX chY     nil)))

    (is (true?   (chEither chX chUnit nil  nil)))
    (is (true?   (chEither chX chUnit nil (X.))))
    (is (false?  (chEither chX chUnit nil (Y.))))
    (is (true?   (chEither chX chY    nil (X.))))
    (is (true?   (chEither chX chY    nil (Y.))))
    (is (false?  (chEither chX chY    nil (Z.))))
    (is (false?  (chEither chX chY    nil nil))))

(testing "(chCompound1 ...)"
    (is                         (chCompound1 (+ 1 2 3 4)))
    (is                         (chCompound1          \c))
    (is                         (chCompound1       "xyz"))
    (is (nil?                   (chCompound1         nil)))
    (is (thrown? AssertionError (chCompound1         3/4)))

    (is (true?                  (chCompound1 nil (+ 1 2 3 4))))
    (is (true?                  (chCompound1 nil          \c)))
    (is (true?                  (chCompound1 nil       "xyz")))
    (is (true?                  (chCompound1 nil         nil)))
    (is (false?                 (chCompound1 nil         3/4))))
```