Iteratees in C

or how to invert the basement plumbing

pesco @khjk.org

HushCon, Seattle, Dec 16 2013

*updated Dec 18 2013

Wat?

- Iteratees are stream processors.
- Programming model / API
  - to allow reasoning about I/O
- Origin: functional programming
- Challenge: Do it without first-class functions!
  - cf. Hammer

About Me

(Context)

- Programming
- Mathematics
- Cryptography

Motivation

- Apply formal thinking
- . . . to program construction
- . . . to achieve security properties.

Iteratees

- Not my idea
- Oleg Kiselyov in Haskell, 2008
- Pretty far-out, even for the Haskellers
  - cf. proliferation of alternative libraries

- A model for input processing
- The model gives rise to a library/implementation.

Iteratees

(cont.)

- Represent stream processing
- Modular, composable
- Breaking procedures over chunks
  - compose without worry of the boundaries
- Inversion of the common representation of input processing
  - "FILE/read $\rightarrow$ file/READ"
  - data as data $\rightarrow$ procedure as data

Security

- High level
- “Declarative”
- Be formal about accepted input
- Modular: reduce unmapped interactions

$\Rightarrow$ Avoid weird machines

- Cf. langsec

How It Works

(in Pseudohaskell)

```
data Iteratee α =
    CONT (Stream → (Stream, Iteratee α))
  | STOP Error
  | DONE (α, Stream)
```

Functional Programming in C

- Lambda expressions
- Partial application
- Lexical closures

vs.

- Lambda lifting
- Environment structs

Functional Programming in C

(cont.)

$$f\,(x\,,y\,,z\,) \;=\; \ldots$$

$$g\,(a\,,xs\,) \;=\; \textbf{map}\;\;(\lambda y \;\rightarrow\; f\,(a\,,y\,,0\,))\;\;xs$$

$$g_{\lambda}\,(a\,) \;=\; \lambda y \;\rightarrow\; f\,(a\,,y\,,0\,)$$
$$g\,(a\,,xs\,) \;=\; \textbf{map}\;\;g_{\lambda}\,(a\,)\;\;xs$$

Functional Programming in C

(cont. 2)

```
int f(int x, int y, int z);

struct Env_gl {int a; /*...*/};
int gl(void *env_, int y) {
    struct Env_gl *env = env_;
    return f(env->a, y, 0);
}

list_t g(int a, list_t xs) {
    struct Env_gl env = {a};
    return map(gl, &env, xs);
}
```

Functional Programming in C

(cont. 2)

```
int f(int x, int y, int z);

struct Env_gl {int a; /*...*/};
int gl(void *env_, int y) {
    struct Env_gl *env = env_;
    return f(env->a, y, 0);
}

list_t g(int a, list_t xs) {
    struct Env_gl env = {a};
    return map(gl, &env, xs);
}
```

Functional Programming in C

(cont. 2)

```
int f(int x, int y, int z);

struct Env_gl {int a; /*...*/};
int gl(void *env_, int y) {
    struct Env_gl *env = env_;
    return f(env->a, y, 0);
}

list_t g(int a, list_t xs) {
    struct Env_gl env = {a};
    return map(gl, &env, xs);
}
```

How It Works

(in C)

```
struct Iteratee_ {
  enum {DONE, CONT, STOP} state;

  union {
    void *result;              // DONE

    struct {                   // CONT,STOP
      Iteratee (*cont)(void *env, Stream *input);
      void *env;
      const char *error;       // STOP only
    };
  };
};
```

How It Works

(in C)

```
struct Iteratee_ {
  enum {DONE, CONT, STOP} state;

  union {
    void *result;              // DONE

    struct {                   // CONT,STOP
      Iteratee (*cont)(void *env, Stream *input);
      void *env;
      const char *error;       // STOP only
    };
  };
};
```

How It Works

(in C)

```
struct Iteratee_ {
  enum {DONE, CONT, STOP} state;

  union {
    void *result;           // DONE

    struct {                // CONT,STOP
      Iteratee (*cont)(void *env, Stream *input);
      void *env;
      const char *error;    // STOP only
    };
  };
};
```

How It Works

(in C)

```
struct Iteratee_ {
  enum {DONE, CONT, STOP} state;

  union {
    void *result;              // DONE

    struct {                   // CONT,STOP
      Iteratee (*cont)(void *env, Stream *input);
      void *env;
      const char *error;  // STOP only
    };
  };
};
```

Case Study: Word Count

```
Iteratee word_ = bind_(dropws, dropword);
Iteratee countwords = wrap(decode(word_), count);

Iteratee it = apply(enumf(stdin), countwords);
uintptr_t nwords = (uintptr_t)finish(it);
```

Case Study: Word Count

```
Iteratee word_ = bind_(dropws, dropword);
Iteratee countwords = wrap(decode(word_), count);

Iteratee it = apply(enumf(stdin), countwords);
uintptr_t nwords = (uintptr_t)finish(it);
```

Case Study: Word Count

```
Iteratee word_ = bind_(dropws, dropword);
Iteratee countwords = wrap(decode(word_), count);

Iteratee it = apply(enumf(stdin), countwords);
uintptr_t nwords = (uintptr_t)finish(it);
```

Case Study: Word Count

```
Iteratee word_ = bind_(dropws, dropword);
Iteratee countwords = wrap(decode(word_), count);

Iteratee it = apply(enumf(stdin), countwords);
uintptr_t nwords = (uintptr_t)finish(it);
```

Case Study: Word Count

(cont.)

- Benchmark: "rockyou" password list
  - 14.344.392 lines
  - ~14.44M words
- `wc -w`
  - 3.8s real 3.6s user 0.1s sys
  - ignores non-ASCII
- `./iter` (`main = test4`)
  - ~~9.2s real 8.5s user 0.7s sys~~
  - ~~total allocation: 600MB~~
  - ~~peak memory use: 4MB~~
  - 3.7s real 3.6s user 0.1s sys
  - total allocation: 3MB

Example: JSON and character encoding

```
Iteratee utf8;  // decodes a single UTF-8 character
Iteratee json;  // consumes Unicode code points
Iteratee it;

it = wrap(decode(utf8), json);  // attach decoder
it = apply(enumf(stdin, it);    // feed input

JSON *j = finish(it);           // extract result
if(j) {
    // ...
}
```

Example: JSON and character encoding

```
Iteratee utf8;  // decodes a single UTF-8 character
Iteratee json;  // consumes Unicode code points
Iteratee it;

it = wrap(decode(utf8), json);  // attach decoder
it = apply(enumf(stdin, it);    // feed input

JSON *j = finish(it);           // extract result
if(j) {
    // ...
}
```

Example: JSON and character encoding

```
Iteratee utf8;  // decodes a single UTF-8 character
Iteratee json;  // consumes Unicode code points
Iteratee it;

it = wrap(decode(utf8), json);  // attach decoder
it = apply(enumf(stdin, it);    // feed input

JSON *j = finish(it);           // extract result
if(j) {
    // ...
}
```

Example: JSON and character encoding

```
Iteratee utf8;  // decodes a single UTF-8 character
Iteratee json;  // consumes Unicode code points
Iteratee it;

it = wrap(decode(utf8), json);  // attach decoder
it = apply(enumf(stdin, it);    // feed input

JSON *j = finish(it);           // extract result
if(j) {
    // ...
}
```

Example: JSON and character encoding

```
Iteratee utf8;  // decodes a single UTF-8 character
Iteratee json;  // consumes Unicode code points
Iteratee it;

it = wrap(decode(utf8), json);  // attach decoder
it = apply(enumf(stdin, it);    // feed input

JSON *j = finish(it);           // extract result
if(j) {
    // ...
}
```

Example: XML and character encoding

- `<?xml>` directive may specify encoding for the file.
  - guess encoding from initial bytes at first.
  - switch to proper decoder for rest of file.
- This is easy with iteratees (`bind`).

$\Rightarrow$ One iteratee to process the entire file

- Also cf. `Content-Type` header, BOM, ...

Example: XML and character encoding

(cont.)

```
Iteratee xml_decl;      // returns encoding name
Iteratee xml_unicode;   // consumes Unicode code points

Iteratee f_xml(void *, void *result) {
  const char *encoding = result;
  Enumeratee decoder = find_decoder(encoding);
  return wrap(decoder, xml_unicode);
}

Iteratee xml = bind(xml_decl, f_xml, NULL);
```

Example: XML and character encoding

(cont.)

```
Iteratee xml_decl;      // returns encoding name
Iteratee xml_unicode;   // consumes Unicode code points

Iteratee f_xml(void *, void *result) {
  const char *encoding = result;
  Enumeratee decoder = find_decoder(encoding);
  return wrap(decoder, xml_unicode);
}

Iteratee xml = bind(xml_decl, f_xml, NULL);
```

Example: XML and character encoding

(cont.)

```
Iteratee xml_decl;      // returns encoding name
Iteratee xml_unicode;   // consumes Unicode code points

Iteratee f_xml(void *, void *result) {
  const char *encoding = result;
  Enumeratee decoder = find_decoder(encoding);
  return wrap(decoder, xml_unicode);
}

Iteratee xml = bind(xml_decl, f_xml, NULL);
```

PoC Implementation

- Basic iteratees
- Input from file descriptor
- "`decode`" combinator
- UTF-8 decoder
- Several simple test examples
  - word count, line count, UTF-8 character count, ...
- Automatic memory management
  - uses standard `malloc`/`free` for arenas
  - x86 (32-bit) only right now (needs to know registers)
- ~1500 lines alltogether

Future Work

- ~~Memory management (garbage collection?!)~~
- Speed optimization
- A larger case study
- Flesh out a proper library/API
- Recursive-descent parser combinators
- Iteratee API for Hammer
- ...

Pointers

- Feedback: pesco@khjk.org
- PoC code: http://code.khjk.org/citer/
- Iteratee home (Oleg):
  http://okmij.org/ftp/Streams.html#iteratee
- Introductory paper (Oleg):
  http://okmij.org/ftp/Haskell/Iteratee/describe.pdf