ソフトウェア基礎科学分野
(住井・松田研究室)

教授　住井 英二郎
准教授　松田 一孝
助教　Oleg Kiselyov

1

教員紹介(住井)

- 1975年　東京生まれ
- 1998年　東京大学 理学部 情報科学科 卒業
- 2000年～2001年　ペンシルバニア大学 客員研究員
- 2001年～2003年　東京大学 助手
- 2003年～2005年　ペンシルバニア大学 RA
- 2010年～現在　日本学術会議（特任）連携会員
- 2005年～現在　東北大学 助教授→准教授→教授
  - 詳しくは http://www.kb.ecei.tohoku.ac.jp/~sumii/ (もしくは短縮URL http://j.mp/SmiThk)を見てください
  - 趣味 (最近): Twitter (@esumii)

2

教員紹介(松田)

- 1982年　愛媛生まれ
- 2004年　東京大学 工学部 計数工学科卒業　(応用数学)
- 2009年　博士（情報理工学）@東大
- 2008年～2010年　学振特別研究員@東大
- 2010年～2012年　東北大学 助教
- 2012年～2015年　東京大学 助教
- 2015年～現在　東北大学 准教授
  - 詳しくは http://www2.sf.ecei.tohoku.ac.jp/~kztk/ (もしくは"松田 一孝 東北大"で検索)

3

教員紹介(Oleg)

Oleg Kiselyov
(オレッグ キセリョーヴ)
http://okmij.org/ftp/

- 1993年　北テキサス大学 計算機科学科 博士課程 修了
- 1996年～2014年　企業→国立研究所 (カルフォルニア)
- 2015年～現在　東北大学 助教

4

研究分野

プログラムとプログラミング言語の
理論と実現
(Theory and Implementation of
Programs and Programming Languages)

- 「プログラム」＝「計算の記述」
- 「プログラミング言語」＝「計算記述体系」
  - C, Java, ML, オートマトン, チューリング機械, λ計算, π計算, ...

5

6

研究室ホームページ兼ブログ
http://www.sf.ecei.tohoku.ac.jp/

興味を持った人は個別見学歓迎！
予約・質問は sf-staff@sf.ecei.tohoku.ac.jp へ

プログラミング言語理論

- 「プログラムを作るときに気をつける」
- 「よく見て確かめる」（レビュー）
- 「試しに動かしてみる」（テスト）etc.

ではなく、論理的基礎

にもとづくアプローチ

※ 両者は対立するものではなく相互補完的

7

研究室の「カリキュラム」

学部3年2月～ (自習)
- 関数型言語OCamlのプログラミング
  (日本語の教科書，練習問題)

学部4年4月～
- プログラミング言語理論の基礎の輪講
  (英語の教科書，定理証明ソフトウェアCoq),
- ゼミ，授業

10月～ ゼミ，卒業研究

3月頃 卒論発表会 (3研究室合同)，できれば学会発表

大学院 (進学の場合)
ゼミ，輪講，授業，修士研究，できれば国際学会発表

8

研究室で学ぶこと

- プログラミング言語理論
- プログラミングを含む情報科学・計算機科学（コンピュータサイエンス）の「常識」
- プログラムを含む一般の物事について、論理的に理解・説明する能力
  – 本来の意味での「コミュニケーション能力」
- 技術的な文章を読み書きする能力（日本語・英語）

9

高度な (≠ 難しい) 最先端理論を 楽しく勉強しましょう！

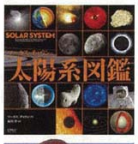
興味のある人は sf-staff@ecei.tohoku.ac.jp まで
メールで予約して、ゆっくりと見学してください

10

Ohmsha
Publisher of Science and Engineering Books

Home
理工学専門書 | コンピュータ・一般書 | 資格試験 | 雑誌
新入荷内 | カートを見る

▸新刊案内
書籍検索
書名　に
上記を含む　検索
詳しく検索 >>
オームビルテナント募集中
eStore (β)
PDF版書籍データ販売
Ohm Bulletin
▸各種セミナーのご案内



専門書店 Books Ohm

Home >コンピュータ・一般書>プログラミング>その他 詳細

型システム入門 プログラミング言語と型の理論

・著者：Benjamin C. Pierce 著
住井英二郎 監訳
遠藤侑介・酒井政裕・今井敬吾・黒木裕介・今井宣洋・才川隆文・今井健男 共訳
・定価：7140円(本体6800円＋税)
・B5　528頁
・ISBN　978-4-274-06911-6
・発売日：2013/03

ツイート 201
いいね！ 21
B! 16
クリップ

カートに入れる
Download　第1章のサンプルダウンロード
訳者によるサポートページ
◆PDF版はこちら
※本体価格は変更される場合があります。
※通常2～3日以内で発送いたします。

[主要目次] [詳細目次]

型システムを理解するうえでの定番書を翻訳

型システムとは、プログラミング言語の安全性や効率を高めるうえで重要な理論・手法です。本書は、その型システムについて基礎的な話題を網羅し、実装例を交えて丁寧に解説したThe MIT Press発行の解説書"Types And Programming Languages"（TAPL）を翻訳したものです。言語設計者や学生だけでなく、静的型付言語を深く理解して活用したいプログラマーにとっても貴重な情報となっています。

★このような方におすすめ
情報科の学生、研究者
静的型付言語を利用するプログラマー

主要目次

http://j.mp/taplTalk

http://j.mp/icfpc2011

ICFP Programming Contest 2011: Official Site

Friday, June 17, 2011

Task description - contest starts now!

[Remark: This post was last updated on 19:00 June 17 Friday UTC. Search it for "Update" on your browser.]

Welcome to the ACM SIGPLAN ICFP Programming Contest 2011! The task this year is to write a program that plays the card game Lambda: the Ultimate Card Game (LTG for short).

Rules

Each match of LTG is played by two programs (player 0 and player 1) in alternate turns (turn 1 of player 0, turn 1 of player 1, turn 2 of player 0, turn 2 of player 1, etc.). Each player owns 256 *slots*, numbered from 0 to 255, and a fixed set of cards. A slot consists of a field and an integer called *vitality*, ranging from -1 to 65535. A field holds a value, which is either an integer, ranging from 0 to 65535, or a function (in the sense of programming languages). A value is a *valid slot number* if it is an integer greater than or equal to 0, and less than or equal to 255. A slot is *dead* if its vitality is 0 or -1; otherwise the slot is *alive*. At the start of each match, every field is initialized to the identity function (a function that takes an argument x and returns x) and every vitality is initialized to 10000. The cards have fixed values as defined below.

In each turn, the player (called the *proponent*) performs either a *left application* or a *right application*. A left application applies (as a function) a card to the field of one of the proponent's slots. A right application applies (as a function) the field of one of the proponent's slots to a card. In either case, an error is raised if the card or the field to be applied is not a function, or if the slot is dead. The other player (called the *opponent*) is able to see which applications the proponent has chosen on what card and slot. [Remark: In mathematics and programming languages (as well as in our rules), one applies a function to an argument, not vice versa, that is, "applying f to x" means "f(x)", not "x(f)".] [Remark: As a result of the rules, all function applications in LTG are "call by value"; that is, the argument x of function application f(x) is always a value.]

The turn ends when an error is raised, when the number of function applications caused by the left or right application (including itself) exceeds 1000, or when the left or right application returns a value without exceeding this limit. In the last case, the field of the slot used for the left or right application is overwritten with the return value. In the other cases, it is overwritten with the identity function. Effects caused by function applications are not reversed and remain even if an error is raised or the application limit is exceeded. Cards are not consumed by applications and can be reused as many times as necessary.

A match ends after 100000 turns of each player, or when every slot of a player has become dead after a turn. In either case, a player wins if it has more slots alive than the other player. The players tie if the numbers of slots alive are equal.

Cards

The set of cards are fixed as follows. The effect of a card is undefined in any case not specified below. [Remark: The images are only for amusement and are not part of the official rules.]

Card "I" is the identity function. [Remark: It is called the I combinator and written λx.x in lambda calculus.]

Blog Archive
▶ 2012 (2)
▼ 2011 (19)
▶ September (2)
▶ July (3)
▼ June (9)
Round 1 finished
Downloading finished and team list published
Contest finished
Task description - contest starts now!
How to prepare an environment for testing your bot
Contest starting in two weeks
▶ May (3)
▶ April (1)
▶ March (1)

Card "zero" is an integer constant 0.

Card "succ" is a function that takes an argument n and returns n+1 if n<65535 (or returns 65535 if n=65535). It raises an error if n is not an integer.

Card "dbl" is a function that takes an argument n and returns n*2 (n times 2) if n<32768 (or returns 65535 if n>=32768). It raises an error if n is not an integer.

Card "get" is a function that takes an argument i and returns the value of the field of the ith slot of the proponent if the slot is alive. It raises an error if i is not a valid slot number or the slot is dead.

Card "put" is a function that takes an (unused) argument and returns the identity function.

Card "S" is a function that takes an argument f and returns another function, which (when applied) will take another argument g and return yet another function, which (when applied) will take yet another argument x, apply f to x

obtaining a return value h (or raise an error if f is not a function), apply g to x obtaining another return value y (or raise an error if g is not a function), apply h to y obtaining yet another return value z (or raise an error if h is not a function), and return z. [Remark: The first function is called the S combinator and written λf.λg.λx.f(x)(g x) in lambda calculus.]

Card "K" is a function that takes an argument x and returns another function, which (when applied) will take another (unused) argument y and return x. [Remark: The first function is called the K combinator and written λx.λy.x in lambda calculus.]

Card "inc" is a function that takes an argument i, and

- increases by 1 the vitality v of the ith slot of the proponent if 0<v<65535,
- does nothing if v=65535 or v<=0, or
- takes an error if i is not a valid slot number,

and returns the identity function.

Card "dec" is a function that takes an argument i, and

- decreases by 1 the vitality v of the (255-i)th slot of the opponent if v>0,
- does nothing if v<=0, or
- raises an error if i is not a valid slot number,

and returns the identity function.

Card "attack" is a function that takes an argument i and returns another function, which (when applied) will take another argument j and return yet another function, which (when applied) will take yet another argument n, decrease by n the vitality v of the ith slot of the proponent (or raise an error if i is not a valid slot number, n is not an integer, or n is greater than v), and

decrease by n*9/10 (n times 9 divided by 10, with the remainder discarded) the vitality w of the (255-j)th slot of the opponent if it is alive (w is set to 0 if it would become less than 0 by this decrease),

- do nothing if the slot is dead, or
- raise an error if j is not a valid slot number,

and return the identity function.

Card "help" is a function that takes an argument i and returns another function, which (when applied) will take another argument j and return yet another function, which (when applied) will take yet another argument n, decrease by n the vitality v of the ith slot of the proponent (or raise an error if i is not a valid slot number, n is not an integer, or n is greater than v), and

- increase by n*11/10 (n times 11 divided by 10, with the remainder discarded) the vitality w of the jth slot of the

proponent if it is alive (w is set to 65535 if it would become greater than 65535 by this increase),

- do nothing if the slot is dead, or
- raise an error if j is not a valid slot number,

and return the identity function.

Card "copy" is a function that takes an argument i, and returns the value of the field of the ith slot of the opponent. It raises an error if i is not a valid slot number. Note that the slot is ith, not (255-i)th.

Card "revive" is a function that takes an argument i, sets to 1 the vitality v of the ith slot of the proponent if v<=0 (or does nothing if v>0), and returns the identity function. It raises an error if i is not a valid slot number.

Card "zombie" is a function that takes an argument i and returns another function, which (when applied) will take another argument x, and

- overwrite with x the field of the (255-i)th slot of the opponent if the slot is dead, or
- raise an error if i is not a valid slot number or the slot is alive,

and set the vitality of the slot to -1, and return the identity function.

Immediately before each turn of a player, the field of every slot of the player with vitality -1 is automatically applied (as a function) to the identity function (even though the slot is dead) in increasing order of slot number. **[Update on 07:05 June 17 Friday UTC: The vitality of each slot is checked just before its automatic application; that is, (1) the vitality of slot 0 is checked, and (2) if it is -1, the field of slot 0 is applied, and then (3) the vitality of slot 1 is checked, and (4) if it is -1, the field of slot 1 is applied, etc.]** For each of these automatic applications, an error is raised if the field is not a function or the number of function applications caused by the application exceeds 1000. An error caused by each automatic application does not affect any other automatic applications. After each automatic application, the field of the slot used for the application is overwritten with the identity function, and the vitality of the slot is reset to 0.

In addition, during the above automatic applications, parts of the effects of 4 cards change from their previous descriptions as follows (the other parts do not change from the original):

- Card "inc" decreases the v (in the previous description of this card) by 1 if v<0, or does nothing if v>=0.
- Card "dec" increases the v by 1 if v>0 and v<65535, or does nothing if v<=0 or v=65535.
- The third function in card "attack" increases the w by n*9/10 if w>0 (w is set to 65535 if it would become greater than 65535 by this increase), or does nothing if w<=0.
- The third function in card "help" decreases the w by n*11/10 if w>0 (w is set to 0 if it would become less than 0 by this decrease), or does nothing if w<=0.

[Remark: For an informational purpose only, executables for interactive LTG plays are provided at:

http://www.sf.ecei.tohoku.ac.jp/icfpc/ltg.linux-32.gz
http://www.sf.ecei.tohoku.ac.jp/icfpc/ltg.linux-64.gz
http://www.sf.ecei.tohoku.ac.jp/icfpc/ltg.macosx-32.exe.tgz
http://www.sf.ecei.tohoku.ac.jp/icfpc/ltg.macosx-32.x-gui.gz (x11 required)]

http://j.mp/icfpc2011video

http://j.mp/itprofun

数理科学的バグ撲滅方法論のすすめ---目次

2007/06/13
ITpro

筆者 住井 英二郎

「プログラミング言語理論」という研究分野がある。この分野の研究者たちは、「ML」「Haskell」「Scheme」あるいは「λ計算」「π計算」(円周率計算のことではない)など、多くのプログラマにとっては聞いたこともない言語やモデルについて、日夜研究している。ただ、そのような言語は「難しい」「役に立たない」などと思われがちだ。

この連載では、こうしたプログラミング言語やソフトウエア科学の様々な研究を、できるだけ普通のプログラマやエンジニアにもわかりやすく(どちらかといえば理論よりも実用に重点をおいて)紹介していく。

更新は毎月第2水曜日(1月のみ第3水曜日)

連載目次

第1回 OCamlを試してみる
第2回 「単一代入」と「末尾再帰」
第3回 計算量の工夫でプログラムは劇的に速くなる
第4回 関数型言語とオブジェクト指向、およびOCamlの"O"について
第5回 LablGLで3Dグラフィックス ～OCamlの「多相バリアント」と「ラベル付き引数」～
第6回 OCamlの「モジュール・システム」
第7回 「代数データ型」でいろいろなデータを表してみる
第8回 独自のプログラミング言語を開発してみよう(その1)
第9回 独自のプログラミング言語を開発してみよう(その2) プログラムを実行できるようにする
第10回 静的スコープと関数クロージャ ～関数型言語のインタプリタを書いてみる～
第11回 クロージャによる超軽量並行プロセスの簡単実装法
第12回 「型推論」の実装法
第13回 「表明」と「契約」による命令型プログラムの形式的検証
第14回 型＝命題、プログラム＝証明
第15回 型からプログラムを当てる

Lightweight Language Future

LL で未来を発明する

セッション概要

「未来を予測する最良の方法は、未来を発明してしまうことだ」アラン・ケイの有名な言葉です。また、ポール・グレアムは「百年の言語 -The Hundred-Year Language」というエッセイを書き、百年後を予想して現在どの選択肢に賭けるべきかを考察しています。

このセッションでは言語設計者、処理系実装者に集まっていただき、次の3つの質問をパネラーにぶつけ、未来の言語、百年後の言語を「発明」していきます。

1. 百年後の言語はどうなっていると思いますか?
2. 1のために、あなたは今まで何をしてきましたか?
3. 1のために、あなたはこれから何をしますか?

出演

- Larry Wall (Perl)
- まつもとゆきひろ(Ruby) (ネットワーク応用通信研究所)
- 住井英二郎(MinCaml) (東北大学)
- 藤田喜勝(Ypsilon) (リトルウィング)
- ひげぽん(Mosh) (サイボウズ・ラボ)

司会: 今泉貴史(千葉大学)

To invent the future by Lightweight Languages

"The best way to predict the future is to invent it," said Alan Kay. Since software will be everywhere in our future, programming languages are #the# tool to build the future. If you want to invent the future, it is crucial to invent the right language.

We invite five panelists from the front line of language design and implementation, and discuss the future of programming languages, starting from asking the following three questions to them:

1. What kind of language do you think will be the best for the next century?
2. What have you done to get the best language?
3. What will you do to invent the best language?

Panelists:

- Larry Wall (Perl)
- Yukihiro "Matz" Matsumoto (Ruby)
- Eijiro Sumii (MinCaml)
- Yoshikatsu Fujita (Ypsilon)
- higepon (Mosh)

Moderator:

- Takashi Imaizumi (Chiba University)

メニュー

開催前トップ
トラックバック
開催のお知らせ
LLFuture×gihyo.jp
ビデオ集1
ビデオ集2
写真集
資料集
プログラム
基調講演
LL で未来を発明する
サイコー？！フレームワーク
LL でアート
キミならどう書く?
古い言語、新しい言語
ライトニング・トーク
チケット販売
公式タグ
ブログ
アクセス
お食事処
乗り換え検索
バナー置き場
開催概要
お世話になった方々
スタッフ紹介
取材をご希望の方へ

Webページ内検索

検索

ブックマーク

公式タグ:llfuture
13 users

ブログ最新記事

LL FutureのWebサイトでもをご覧になれます
LL Golf Hole 9
当日のお忘れ物について
OP.ED ビデオ、各種資料
トラックバックをお待ちしています

http://j.mp/llfuture

2015-09-07

ICFP 2015

教授の住井、准教授の松田さん、助教のオレッグさん、ならびに修士2年の徳田くんが、カナダのバンクーバーで開かれたICFP 2015（関数型プログラミングに関する国際学会）に参加しました。写真は松田さんの発表です（動画）。徳田くんの発表の動画もあります。来年のICFP 2016は日本の奈良で開かれる予定です（住井がプログラム委員長を拝命しました。よろしくお願いします）。

PREV NEXT

- Hello, you're not logged in. Login now!
- Shopping Cart (0 Articles : $ 0.00)
- Help?

- Shop
- Designs
- Purchase
- Help ?

This shop requires cookies in order to work properly. Please click here to activate cookies.

All products

Product Details

 Front

 Back

 Right

 Left

Oleg Already Did It

Men's T-Shirt

Classic-cut standard weight t-shirt for men, 100% pre-shrunk cotton, Brand: Gildan

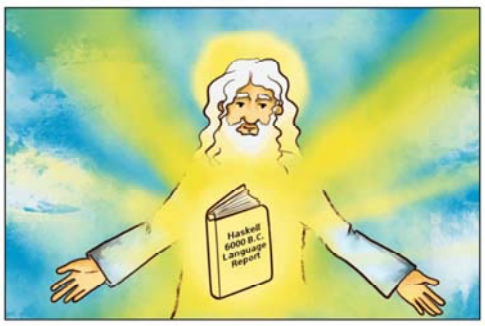
Details

Oleg Kiselyov, computer scientist, already solved that. In the type system.

Cartesian Closed Comic

Iteratees

=<<<<<id>>>>>=



God created Haskell: and everything in Haskell was lazy*, including IO: and God liked it.

* non-strict

People said to each other, «Come, let's make libraries and bake them thoroughly.»
Then they said, «Come, let us build ourselves an iteratee IO library.
Otherwise we will suffer from the shortcomings of lazy IO.»

Cartesian Closed Comic #15: Iteratees https://ro-che.info/ccc/15

God said, «If they can invent their own IO, then nothing they plan to do will be impossible for them. Come, let us go down and confuse their iteratee libraries so they will not understand each other.»

The variety of streaming IO libraries confused developers, and everyone continued to use lazy IO.

See e.g. this discussion about the leftovers problem.

=<<<<<id>>>>>=
ArchiveSubscribeAuthors
Published on October 20, 2012

Cartesian Closed Comic #15: Iteratees https://ro-che.info/ccc/15

Iteratee

From Wikipedia, the free encyclopedia

In functional programming, an **iteratee** is a composable abstraction for incrementally processing sequentially presented chunks of input data in a purely functional fashion. With iteratees, it is possible to lazily transform how a resource will emit data, for example, by converting each chunk of the input to uppercase as they are retrieved or by limiting the data to only the five first chunks without loading the whole input data into memory. Iteratees are also responsible for opening and closing resources, providing predictable resource management.

On each step, an iteratee is presented with one of three possible types of values: the next chunk of data, a value to indicate no data is available, or a value to indicate the iteration process has finished. It may return one of three possible types of values, to indicate to the caller what should be done next: one that means "stop" (and contains the final return value), one that means "continue" (and specifies how to continue), and one that means "signal an error". The latter types of values in effect represent the possible "states" of an iteratee. An iteratee would typically start in the "continue" state.

Iteratees are used in Haskell and Scala (in the Play Framework[1] and in Scalaz), and are also available for F#.[2] Various slightly different implementations of iteratees exist. For example, in the Play framework, they involve Futures so that asynchronous processing can be performed.

Because iteratees are called by other code which feeds them with data, they are an example of inversion of control. However, unlike many other examples of inversion of control such as SAX XML parsing, the iteratee retains a limited amount of control over the process. It cannot reverse back and look at previous data (unless it stores that data internally), but it can stop the process cleanly without throwing an exception (using exceptions as a means of control flow, rather than to signal an exceptional event, is often frowned upon by programmers[3]).

Contents

Commonly associated abstractions

The following abstractions are not strictly speaking necessary to work with iteratees, but they do make it more convenient.

Enumerators

An **Enumerator** (not to be confused with Java's Enumeration interface) is a convenient abstraction for feeding data into an iteratee from an arbitrary data source. Typically the enumerator will take care of any necessary resource cleanup associated with the data source. Because the enumerator knows exactly when the iteratee has finished reading data, it will do the resource cleanup (such as closing a file) at exactly the right time – neither too early nor too late. However, it can do this without needing to know about, or being co-located to, the implementation of the iteratee – so enumerators and iteratees form an example of separation of concerns.

Enumeratees

An **Enumeratee** is a convenient abstraction for transforming the output of either an enumerator or iteratee, and feeding that output to an iteratee. For example, a "map" enumeratee would map a function over each input chunk.[3]

Motivations

Iteratees were created due to problems with existing purely functional solutions to the problem of making input/output composable yet correct. Lazy I/O in Haskell allowed pure functions to operate on data on disk as if it were in memory, without explicitly doing I/O at all after opening the file - a kind of memory-mapped file feature - but because it was impossible in general (due to the Halting problem) for the runtime to know whether the file or other resource was still needed, excessive numbers of files could be left open unnecessarily, resulting in file descriptor exhaustion at the operating system level. Traditional C-style I/O, on the other hand, was too low-level and required the developer to be concerned with low-level details such as the current position in the file, which hindered composability. Iteratees and enumerators combine the high-level functional programming benefits of lazy I/O, with the ability to control resources and low-level details where necessary afforded by C-style I/O.[4]

Examples

Uses

Iteratees are used in the Play framework to push data out to long-running Comet and WebSocket connections to web browsers.

Iteratees may also be used to perform incremental parsing (that is, parsing that does not read all the data into memory at once), for example of JSON.[5]

It is important to note, however, that iteratees are a very general abstraction and can be used for arbitrary kinds of sequential information processing (or mixed sequential/random-access processing) - and need not involve any I/O at all. This makes it easy to repurpose an iteratee to work on an in-memory dataset instead of data flowing in from the network.

History

In a sense, a distant predecessor of the notion of an enumerator pushing data into a chain of one or more iteratees, was the pipeline concept in operating systems. However, unlike a typical pipeline, iteratees are not separate processes (and hence do not have the overhead of IPC) - or even separate threads, although they can perform work in a similar manner to a chain of worker

threads sending messages to each other. This means that iteratees are more lightweight than processes or threads - unlike the situations with separate processes or threads, no extra stacks are needed.

Iteratees and enumerators were invented by Oleg Kiselyov for use in Haskell.[4] Later, they were introduced into Scalaz (in version 5.0; enumeratees were absent and were introduced in Scalaz 7) and into Play Framework 2.0.

Formal semantics

Iteratees have been formally modelled as free monads, allowing equational laws to be validated, and employed to optimise programs using iteratees.[4]

Alternatives

- Iterators may be used instead of iteratees in Scala, but they are imperative, so are not a purely functional solution.
- In Haskell, two alternative abstractions known as Conduits and Pipes have been developed. (These Pipes are not operating system level pipes, so like iteratees they do not require the use of system calls).
- There is also a high-level abstraction named Machines (http://hackage.haskell.org/package/machines) (implemented in Scala on top of Scalaz as scalaz-stream (https://github.com/scalaz/scalaz-stream)).
- In Haskell, the package safe-lazy-io (http://hackage.haskell.org/cgi-bin/hackage-scripts/package/safe-lazy-io) exists. It provides a simpler solution to some of the same problems, which essentially involves being "strict enough" to pull all data that is required, or might be required, through a pipeline which takes care of cleaning up the resources on completion.

References

1. "Handling data streams reactively". *Play Framework documentation*. Retrieved 29 June 2013.
2. "Github Search Results: Iteratee in FSharpx".
3. "Java theory and practice: The exceptions debate". *IBM developerWorks*. Retrieved 17 May 2014. **Cite error: Invalid <ref> tag; name "play-enumeratee" defined multiple times with different content (see the help page).**
4. Kiselyov, O. (2012). "Iteratees". *Functional and Logic Programming*. Lecture Notes in Computer Science **7294**. pp. 166–181. doi:10.1007/978-3-642-29822-6_15. ISBN 978-3-642-29821-9.
5. James Roper (10 December 2012). "Json.scala". *play-iteratees-extras*. Retrieved 29 June 2013.

Further reading

- John W. Lato (12 May 2010). "Iteratee: Teaching an Old Fold New Tricks". *Issue #16 of The Monad Reader*. Retrieved 29 June 2013. This relates to Haskell.

External links

- **Scala tutorials**
  - **Play 2.0**
    - Understanding Play 2 iteratees for normal humans (http://mandubian.com/2012/08/27/understanding-play2-iteratees-for-normal-humans/)
    - Iteratees for imperative programmers (http://jazzy.id.au/default/2012/11

/06/iteratees_for_imperative_programmers.html)
  - **Scalaz**
    - Scalaz tutorial: Enumeration-based I/O with iteratees (http://blog.higher-order.com/blog/2010/10/14/scalaz-tutorial-enumeration-based-io-with-iteratees/)
- **Haskell tutorials**
  - Stanford lecture notes (http://www.scs.stanford.edu/11au-cs240h/notes/iteratee.html)
- **Further information**
  - Oleg Kiselyov's Iteratees and Enumerators page (http://okmij.org/ftp/Streams.html)

Categories: Functional programming | Iteration in programming