Recaffeinating Java

Modular and Concise Semantic Polymorphism for Java

Please
anonymize
your paper

Please
anonymize
your paper

Abstract

To do (??)

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

Over the years, many programming language extensibility mechanisms have been proposed for interesting programming models (e.g., asynchronous programming [1], actor-based concurrency, parallel programming, workflows, database queries [7]). F# *computation expressions* [11], macros, Haskell Embedded DSLs (shallow/deep, do-notation for monads) are mechanisms that create a familiar look–and–feel syntactically without fixing the semantics of written code. Currently, vanilla Java lacks such kind of lightweight extensibility.

In this work we introduce such syntactic flexibility using *Object Algebras* [10] (hereafter **OAs**) as a device for the description of the semantics. OAs are simple to understand, they solve the expression problem, they have attracted a lot of attention in the research community in the area of interpreters and have a simple model of development. Overall, this will achieve an elegant and concise way in Java to author libraries that affect the semantics of code. The motivating observation for the current proposal is that F#'s builders of computation expressions [16, Chapter 6.3.10] have an 1-1 correspondence with the signature of an OA.

2. Background

To do (??) Our work is inspired by F#'s computation expressions. Async, AsyncSeq, Maybe, Cloud and many more define different semantics. the translation is made during type checking phase: can be characterised as type driven as shown but seems as a syntactic translation. F# is monadic + operators (Bind + Return are monadic). Seq isn't build with this pattern though.

To do (??)

In F# asynchronous workflows are designated with the `async` label which corresponds to a computation expression defined in the standard library e.g.,

```
let getLength url = async {
  let! html = fetchAsync url
  do! Async.Sleep 1000
  return html.Length
}
```

The corresponding translation is type-driven:

```
async.Bind(fetchAsync(url), fun html ->
  async.Bind(Async.Sleep 1000, fun () ->
    async.Return(html.Length)))
```

3. Java transformation into CPS

3.1 Delimiting transformation at the method level

4. Embedded DSLs

4.1 Async

Recursive asynchronous fibonacci.

4.2 Yield with exceptions

A pull-based stream library.

4.3 Generating HTML

To do (??)

5. Related Work

We follow a similar approach with scala-virtualized [9] although we perform our translation via a separate rewriting tool.

Implementing First-Class Polymorphic Delimited Continuations by a Type-Directed Selective CPS-Transform [12]

Vazou et al, in From Monads to Effects and Back [17] rely on the theory of Representing Monads [4] and represent everything as the continuation monad. We follow a similar approach based on the translation of Dart as described in [8]. Lightweight Monadic Programming [14] follows a type driven approach. Tagless Interpreters [2]

6. Future Work

That *Monads do not compose* has been identified in Steele's paper [13]. There are three approaches to solve this problem (as identified in [5]): monad transformers [6], the free monad as used in [15] and side–effect–request handlers [3]. One interesting research question is to study the differences between these three and identify early on, if we can borrow one technique and apply it to our design.

References

[1] G. Bierman, C. Russo, G. Mainland, E. Meijer, and M. Torgersen. Pause 'N' Play: Formalizing Asynchronous C#. In *ECOOP 2012*.

[2] J. Carette, O. Kiselyov, and C.-C. Shan. Finally tagless, partially evaluated. In *Programming Languages and Systems*.

[3] R. Cartwright and M. Felleisen. Extensible denotational language specifications. In *TACS 1994*.

[4] A. Filinski. Representing monads. In *POPL 1994*.

[5] O. Kiselyov and H. Ishii. Freer Monads, More Extensible Effects. In *Haskell 2015*.

[6] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL 1995*.

[7] E. Meijer, B. Beckman, and G. Bierman. LINQ: reconciling object, relations and xml in the. net framework. In *SIGMOD 2006*.

[8] E. Meijer, K. Millikin, and G. Bracha. Spicing up dart with side effects. *Queue*, 13(3):40:40–40:59, Mar. 2015.

[9] A. Moors, T. Rompf, P. Haller, and M. Odersky. Scala-virtualized. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*, PEPM '12, pages 117–120, New York, NY, USA, 2012. ACM.

[10] B. C. d. S. Oliveira and W. R. Cook. Extensibility for the Masses. In *ECOOP 2012*.

[11] T. Petricek and D. Syme. The F# Computation Expression Zoo. In *PADL 2014*.

[12] T. Rompf, I. Maier, and M. Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. *SIGPLAN Not.*, 44(9):317–328, Aug. 2009.

[13] G. L. Steele and Jr. Building Interpreters by Composing Monads. In *POPL 1994*.

[14] N. Swamy, N. Guts, D. Leijen, and M. Hicks. Lightweight monadic programming in ML. In *ICFP 2011*.

[15] W. Swierstra. Data Types à La Carte. *J. Funct. Program.*, pages 423–436, July 2008.

[16] D. Syme. The F# 3.0 Language Specification, Sept. 2012.

[17] N. Vazou and D. Leijen. From monads to effects and back. In M. Gavanelli and J. Reppy, editors, *Practical Aspects of Declarative Languages*, volume 9585 of *Lecture Notes in Computer Science*, pages 169–186. Springer International Publishing, 2016.