Broken Links (Presentation)

Frank Emrich
The University of Edinburgh
United Kingdom
frank.emrich@ed.ac.uk

Daniel Hillerström
The University of Edinburgh
United Kingdom
daniel.hillerstrom@ed.ac.uk

Abstract

Links is a feature-rich web programming language for research. It has often been the case that new features have been designed and studied in isolation. Unsurprisingly, this 'methodology' inevitably leads to a system in which the interaction amongst several features is broken. However, for some combination of features it is not necessarily clear how to make them apt together. In this talk, we want to give a demonstration of the capabilities of the Links web programming system followed by a discussion of interesting open research challenges that have arisen from our work on Links.

Keywords multi-tier web programming, linearity, session types, effect handlers

1 Introduction

An ever-growing set of tools and technologies help web developers write increasingly more interactive and dynamic web applications. As a consequence, the complexity of web applications has sky-rocketed. The programmer must master a multitude of technologies as a single web application may comprise several heterogeneous programming technologies split across multiple tiers: the client may be written in JavaScript, communicating in ad-hoc manner with an application server which may be a containerised distributed system deployed on a cloud cluster, querying and crunching data on multiple SQL-flavoured database servers. This is an instance of the *impedance mismatch problem*. An old problem that has been relevant for the Web since its inception.

The Links programming language was originally designed in the mid-2000s to ease the impedance mismatch for web applications by providing a single source language for each involved tier [10]. The name 'Links' is both the designation for the programming language and its associated system. The system automatically slices any given source program, generating the necessary code for each tier; for example translating some code into JavaScript for the browser, some into bytecode for the server, and some into SQL for the database. The system also encompasses an application server, which makes it simple to deploy a Links web application.

Links has been a successful vehicle for programming language research [4, 5, 7–12, 14, 16–29, 31–33, 38, 39, 41]; for example it led to the work of formlets [10, 11] as an idiom for abstracting web forms. Furthermore, Links has influenced other successful web programming languages, including Opa [2] and Ur/Web [6].

2 The Links Language

Over the course of time, the programming language Links has evolved into a feature-rich strict functional language in the spirit of ML [35] with an advanced structural type-and-effect system based on rows [32, 37]; it supports first-class polymorphism with complete type inference for the decidable fragment of the type system [15]. The system also supports linear types. The type-and-effect system is used to classify which code can run on the database, server, or client [4]. Database queries are integrated into the language [9], as pioneered by Kleisli [3, 40] and popularised by LINQ [34], with support for data provenance [16–19] and view update via relational lenses [31].

Inspired by Erlang [1], Links provides lightweight and scalable concurrency, enabling programmers to modularise their code into cohesive units, which communicate asynchronously via linear session-typed channels. Such session types [30] provide a lightweight means for enforcing communication protocol conformance at compile-time. In Links, protocol conformance checking leverages the linear type system to ensure that each channel is used exactly once. Session-typed channels are also failure-safe, meaning communication protocols are capable of accounting for failure [20, 21]. This makes it viable to implement a session-typed distributed application in Links, which can cope with client drops.

Links has native support for effect handlers [36] as a means for structuring effectful computations [23–29]. Effect handlers generalise exception handlers as they let the programmer obtain the continuation at the throw point of an operation. This continuation is delimited by the nearest suitable enclosing handler. Furthermore, the continuation is a first-class entity which can be resumed immediately, stashed for later, or discarded. Handlers enable expressive control abstractions to be implemented as user-definable libraries. For instance, the idiom async/await for asynchronous programming is definable in user code using handlers [13].

3 Challenging Interactions

We outline three different research challenges that we have discovered during our implementation work on Links.

Combining Handlers and Sessions Session types enjoy strong meta-theoretic properties such as *deadlock-freedom* – which holds true in Links too, if one disregards the effect handler fragment of the language.

We provide a minimal example to to illustrate how one might break the deadlock property for session types using effect handlers. The idea is to take two processes, a sender and a receiver, which communicate over a session-typed channel that expects to send (dually receive) one integer. In the receiver process we perform an effectful operation to obtain the continuation prior to the first receive. By invoking this continuation twice we can receive twice from the same linear channel. The second receive deadlocks the program. We may implement the sender process as follows.

```
sig sp : (!Int.End) -> ()
fun sp(oc) { discard(send(42, oc)) }
```

This declares a function `sp` which takes as input a session-typed channel, that expects a single integer to be sent along it. The definition of `sp` takes the output channel `oc` as its sole argument and in its body it sends `42` along the channel, and subsequently discards the continuation-channel (the `End` part). We may realise the receiver process as follows.

```
sig rp : (?Int.End) {Yield:()}-> ()
fun rp(ic) { do Yield; discard(recv(ic).2) }
```

The signature of `rp` states that it takes an input channel which expects one integer. The prefix on the function arrow is an effect row. It describes which effectful operations the function may perform. In this instance it may perform a nullary operation `Yield`, which returns a value of type unit. In the body of `rp` we first perform the operation `Yield` using the **`do`**-invocation form. Afterwards, we perform a receive over the channel `ic`, which returns a pair consisting of the integer and the continuation channel. We project the latter and discard it.

Finally, we connect the two processes and give an interpretation of the operation `Yield` using a handler as follows.

```
handle(rp(fork(sp))) {
  case Yield(resume) -> resume(()); resume(())
}
```

The `fork` function provides the sender process with an output channel and returns an input channel which we give to the receiver process. Both processes run under a handler, which intercepts and interprets invocations of `Yield`. Once the receiver process performs `Yield` control gets transferred to the `Yield`-clause in the handler.The clause provides access to the continuation of the invocation, here named `resume`. In the right hand side of the clause, the first invocation of `resume` transfers control back to the receiver process, which then performs the receive which blocks until the sender process has sent its data. Both processes complete after having successfully sent and received the data, respectively. Upon completion control is transferred back into the handler clause, causing the next invocation of `resume`, which transfers control back into `rp`, causing a second receive to happen along the input channel. But no further input is coming along this channel, hence the process blocks indefinitely.

The problem is that the interpretation of `Yield` is non-linear in this example. To conform with the session protocol the continuation `resume` must be invoked exactly once. The pressing question is: *how do we enforce a linear interpretation?*

Handling without Tiers Functions can be located on either a client, server, or database. A server-side function may call a client-side function, by passing the static name of the said function to the client and await the result. However, with effect handlers it is possible to install a handler on the server, and run an effectful client-side computation under it. The continuation of an operation invocation is dynamic and nameless, making it troublesome to pass it back to the server for handling. A solution may be to serialise the entire continuation, however, that may not be feasible for large computations. Another solution may be to generate dynamic names, or pointers, for continuations and storing them on the client. However, the server may drop the provided continuation, thus this approach seems to invite a whole range of lifetime and potential memory leak issues.

ML Type Inference, Linearity, and Kinding The type system supports linear typing, effect tracking, and first-class polymorphism. The addition of type and kind inference leads to intricate interactions between these features.

For example, we utilise the kind system to track linearity of type variables. To make type inference tractable in the presence of first-class polymorphism we also use the kind system to track whether any given type variable is monomorphic. The type inference algorithm uses some bidirectionality to decide when to infer polymorphic types. If a function is annotated with a type signature, then we push the type information of its parameters inwards during inference of its body type. For an unannotated function, we create a fresh type variable for each of its parameters. Each fresh type variable is given the kind `mono`, which prohibits the variable from being unified with a quantified type. This is at odds with inference for linearity. For example, what should the inferred type of **`fun`** `id(x){x}` be? Should `x` be assigned a type variable with the linear kind or the monomorphism kind?

4 Talk Objectives

The talk will cover: i) a demonstration of the Links web programming system, ii) motivated examples of fragile interaction between distinct features, iii) a discussion about research questions arising from our implementation work.

Acknowledgments

We would like to thank Sam Lindley and Leo White for insightful discussions about this work. Daniel Hillerström was supported by EPSRC grant EP/L01503X/1 (EPSRC Centre for Doctoral Training in Pervasive Parallelism).

References

[1] Joe Armstrong. 2010. Erlang. *Commun. ACM* 53, 9 (2010), 68–75.

[2] Henri Binsztok. 2011. The Opa Language. http://opalang.org.

[3] Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. 1995. Principles of Programming with Complex Objects and Collection Types. *Theor. Comput. Sci.* 149, 1 (1995), 3–48.

[4] James Cheney, Sam Lindley, Gabriel Radanne, and Philip Wadler. 2014. Effective quotation: relating approaches to language-integrated query. In *PEPM*. ACM, 15–26.

[5] James Cheney, Sam Lindley, and Philip Wadler. 2013. A practical theory of language-integrated query. In *ICFP*. ACM, 403–416.

[6] Adam Chlipala. 2015. Ur/Web: A Simple Model for Programming the Web. In *POPL*. ACM, 153–165.

[7] Chi-Feng Chou. 2011. *Functional reactive animation in SVG for the web via Links.* Master's thesis. The University of Edinburgh.

[8] Ezra Cooper. 2009. *Programming Language Features for Web Application Development.* Ph.D. Dissertation. The University of Edinburgh.

[9] Ezra Cooper. 2009. The Script-Writer's Dream: How to Write Great SQL in Your Own Language, and Be Sure It Will Succeed. In *Database Programming Languages*. Springer Berlin Heidelberg, Berlin, Heidelberg, 36–51.

[10] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: Web Programming Without Tiers. In *FMCO (LNCS)*, Vol. 4709. Springer, 266–296.

[11] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2008. The Essence of Form Abstraction. In *APLAS (Lecture Notes in Computer Science)*, Vol. 5356. Springer, 205–220.

[12] Ravi Shankar Dangeti. 2008. *Building biological database applications using Links.* Master's thesis. The University of Edinburgh.

[13] Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, K. C. Sivaramakrishnan, and Leo White. 2017. Concurrent System Programming with Effect Handlers. In *TFP (Lecture Notes in Computer Science)*, Vol. 10788. Springer, 98–117.

[14] Gilles Dubochet. 2005. *The SLinks language.* Master's thesis. The University of Edinburgh.

[15] Frank Emrich, Sam Lindley, Jan Stolarek, James Cheney, and Jonathan Coates. 2020. FreezeML: Complete and easy type inference for first-class polymorphism. Draft.

[16] Stefan Fehrenbach. 2019. *Language-integrated Provenance.* Ph.D. Dissertation. The University of Edinburgh.

[17] Stefan Fehrenbach and James Cheney. 2015. Language-integrated Provenance in Links. In *TaPP*. USENIX Association.

[18] Stefan Fehrenbach and James Cheney. 2018. Language-integrated provenance. *Sci. Comput. Program.* 155 (2018), 103–145.

[19] Stefan Fehrenbach and James Cheney. 2019. Language-integrated provenance by trace analysis. In *DBPL*. ACM, 74–84.

[20] Simon Fowler. 2019. *Typed Concurrent Functional Programming with Channels, Actors, and Sessions.* Ph.D. Dissertation. The University of Edinburgh.

[21] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. 2019. Exceptional asynchronous session types: session types without tiers. *PACMPL* 3, POPL (2019), 28:1–28:29.

[22] Simon Fowler, Sam Lindley, and Philip Wadler. 2017. Mixing Metaphors: Actors as Channels and Channels as Actors. In *ECOOP (LIPIcs)*, Vol. 74. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 11:1–11:28.

[23] Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *TyDe@ICFP*. ACM, 15–27.

[24] Daniel Hillerström and Sam Lindley. 2018. Shallow Effect Handlers. In *APLAS (Lecture Notes in Computer Science)*, Vol. 11275. Springer, 415–435.

[25] Daniel Hillerström, Sam Lindley, and Robert Atkey. 2020. Effect Handlers via Generalised Continuations. J. Funct. Program.. To appear.

[26] Daniel Hillerström, Sam Lindley, Robert Atkey, and KC Sivaramakrishnan. 2017. Continuation Passing Style for Effect Handlers. In *FSCD (LIPIcs)*, Vol. 84. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 18:1–18:19.

[27] Daniel Hillerström. 2015. *Handlers for Algebraic Effects in Links.* Master's thesis. The University of Edinburgh.

[28] Daniel Hillerström. 2016. *Compilation of Effect Handlers and their Applications in Concurrency.* Master's thesis. The University of Edinburgh.

[29] Daniel Hillerström, Sam Lindley, and KC Sivaramakrishnan. 2016. Compiling Links Effect Handlers to the OCaml Backend. ML Workshop.

[30] Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR (Lecture Notes in Computer Science)*, Vol. 715. Springer, 509–523.

[31] Rudi Horn, Roly Perera, and James Cheney. 2018. Incremental Relational Lenses. *Proc. ACM Program. Lang.* 2, ICFP, Article Article 74 (July 2018), 30 pages. https://doi.org/10.1145/3236769

[32] Sam Lindley and James Cheney. 2012. Row-based effect types for database integration. In *TLDI*. ACM, 91–102.

[33] Sam Lindley and J Garrett Morris. 2017. Lightweight functional session types. *Behavioural Types: from Theory to Tools. River Publishers* (2017), 265–286.

[34] Microsoft Corporation. 2005. DLinq: .NET Language Integrated Query for Relational Data.

[35] Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. 1997. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA.

[36] Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (2013).

[37] Didier Rémy. 1994. Theoretical Aspects of Object-oriented Programming. MIT Press, Cambridge, MA, USA, Chapter Type Inference for Records in Natural Extension of ML, 67–95.

[38] Gabriel Tellez. 2008. *Implementing the Java Pet Store in Links: An assessment of Links as an effective platform for building web applications.* Master's thesis. The University of Edinburgh.

[39] Thomas Weber. 2018. *Uselets: UIs using Actors as an Abstraction for Composable Communicating Components.* Master's thesis. LMU/University of Augsburg/TU Munich.

[40] Limsoon Wong. 2000. Kleisli, a functional query system. *J. Funct. Program.* 10, 1 (2000), 19–56.

[41] Jeremy Yallop. 2010. *Abstraction for web programming.* Ph.D. Dissertation. The University of Edinburgh.