Thesis for the degree of Doctor of Philosophy

Functional EDSLs for Web Applications

Anton Ekblad

CHALMERS

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2018

Functional EDSLs for Web Applications
Anton Ekblad

ISBN 978-91-7597-692-1
Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 4373
ISSN 0346-718X

Technical report 154D
Department of Computer Science and Engineering
Functional Programming Division

Chalmers University of Technology
SE-412 96 Gothenburg, Sweden
Telephone +46 (0)31-772 10 00

Printed at Chalmers
Gothenburg, Sweden 2018

Functional EDSLs for Web Applications

Anton Ekblad
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

This thesis aims to make the development of complex web applications easier, faster and safer through the application of strongly typed functional programming techniques.

Traditional web applications are commonly written in the de facto standard language of the web, JavaScript, which, being untyped, provides no guarantees regarding the data processed by programs, increasing the burden of testing and defensive programming.

Modern web applications are often highly complex, with multiple interdependent parts interacting over the Internet. Such applications are traditionally implemented with each component as a separate program, exposing its functionality to other components through different API:s over some communication protocol such as HTTP.

This process is mostly manual, and thus error-prone and labour intensive, with accidental API incompatibility between components being particularly problematic. Even in a conventional typed language, the absence of such incompatibilities is not guaranteed. While the different components may well be type-safe in isolation, there is no guarantee that the whole is type-safe as the communication between components is not type-checked.

We present a web application development framework, based on the Haskell programming language, to increase programmer productivity and software quality by addressing these issues. In our framework, an application with an arbitrary number of components is written, compiled and type-checked as a single program, guaranteeing that the application as a whole, including network communication, is type-safe. Communication between components is automatically generated by our framework, eliminating the risk of API incompatibilities completely.

Supporting this framework, we also present a state-of-the-art compiler from Haskell to JavaScript, a novel foreign function interface to allow programs to leverage existing JavaScript code, an embedded language for integrating low-level, high-performance kernels into otherwise high-level web applications, and a highly expressive relational database language.

Keywords: web applications, distributed systems, functional programming, domain-specific programming languages, tierless programming languages

This thesis is based on the work contained in the following papers:

I. A. Ekblad. 2018. Internals of the Haste Compiler.
Preprint. https://ekblad.cc/pubs/selda.pdf

II. A. Ekblad. 2015. Foreign Exchange at Low, Low Rates. Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages, 2, 2015. ACM.

III. A. Ekblad. and K. Claessen. 2014. A Seamless, Client-Centric Programming Model for Type-Safe Web Applications. In Proceedings of the 2014 ACM SIGPLAN International Symposium on Haskell. ACM.

IV. A. Ekblad. 2017. A Meta-EDSL for Distributed Web Applications. In Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell. ACM.

V. A. Ekblad. 2016. High-Performance Web Applications through Haskell EDSLs. In Proceedings of the 9th ACM SIGPLAN International Symposium on Haskell. ACM.

VI. A. Ekblad. 2017. Scoping Monadic Relational Database Queries. Preprint. https://ekblad.cc/pubs/selda.pdf

With the exception of Paper III, all work presented in this thesis was conceived, carried out and documented solely by the author.

In the case of Paper III, the problem statement, plus revision work and feedback on the paper itself, was provided by Koen Claessen, whereas the design, implementation and evaluation of the programming model, as well as the writing itself, was carried out by the author.

Contents

Acknowledgements

There are many, many extraordinary individuals who deserve to be credited for their awesome contributions to these past five years of my life and, by extension, to this thesis. It's been a fantastic ride – if slightly bumpy at times – mainly because I've had such a great team to share it with.

First and foremost, I'd like to thank my amazing supervisor Koen Claessen. For your continuous support, for believing in me even when I didn't myself, for encouraging me to chase my ideas, for inspiring me to embark on this journey in the first place, and – last but not least – for all the fun discussions we've had.

To my awesome co-supervisors Emil Axelsson and Alejandro Russo; thank you for your great support and advice!

To Dimitrios Vytiniotis, my supervisor during three intense months at Microsoft Research; for your guidance, for the opportunity to be a part of the Ziria project, and for your invaluable support when I needed it the most – thank you!

To Christine Räisänen and Gerardo Schneider; without your timely advice, this thesis would in all likelihood not exist. To all my colleagues; thank you for making Chalmers such a fun and rewarding place to work. I will miss you all!

To my life partner, Sofia Zaid, and to my other family and friends; your support is the foundation upon which this thesis was built.

Introduction

Functional EDSLs for Web Applications

This thesis explores the application of strongly typed functional programming and embedded language techniques to the domain of distributed web applications. By leveraging the existing ecosystem of the *Haskell* programming language in the creation and popularisation of novel programming models, we hope to improve the safety, ease of development and time to market of modern, distributed web applications.

1 Introduction

In the production of this thesis, no less than three web applications were used for feedback and communication, two for compiling the bibliography, and another three for handling the paperwork. In fact, most of us make extensive use of web applications to manage most aspects of our lives.

It is easy to see the appeal of a web application over a conventional desktop application, both from a user and a developer perspective. For the user, web applications don't require installation, only requiring the user to point their browser to an address and supply their credentials. Connectivity is included by default, and the user's data travels with them from device to device. For the developer, being able to code against a (relatively) homogeneous browser environment is vastly preferable to testing a wide variety of hardware and operating system combinations. Any number of platforms can be supported as long as each has a web browser, and per-user accounts for an online service make it easy to prevent piracy and divide the service into separate versions with different pricing models.

Anatomy of a web application A typical web application consists of three *tiers*: frontend, backend and database. The exact division of tasks between tiers varies considerably depending on the application's problem domain, requirements for latency or offline operation, and other factors. As a rough guideline, however, the frontend mainly interfaces with the user, the backend deals with authentication, communication and computation, and the database deals with data storage and retrieval.

Traditionally, web applications have been built in a monolithic manner: a single client program talking to a single server program over HTTP, using some ad hoc API devised specifically for the application. More recently, a design methodology known as *microservices* has gained popularity: breaking the server monolith up into multiple small services according to the single responsibility principle. The client may then communicate with any number of these small services, and the services may of course also communicate with each other [Namiot and Sneps-Sneppe, 2014].

Common to both methodologies is that the client and the server(s) are treated as independent applications who only happen to be talking to each other. For better or for worse, this property means that web applications are often less cohesive than their more conventional counterparts. That the three tiers of a web application are commonly implemented using different languages makes this divide even wider.

For applications completely or partially implemented using typed programming languages in particular, this has far-reaching implications: while client and servers may well be type-safe in isolation, the application as a whole is not, as its constituent parts are all built and type-checked separately from each other. Problems which can be seen as type mismatches, which in theory could have been caught by a compiler with relative ease, thus instead morph into runtime errors in the communication code.

Tierless web languages To remedy this situation and improve the type-safety of web applications, a multitude of so-called *tierless languages*, such as *Opa* [Rajchenbach-Teller, 2010], *Ur* [Chlipala, 2010], and *Links* [Cooper et al., 2007], were devised. These languages allow developers to implement a web application as a single program, using the same language for all tiers and extending the guarantees provided by the type checker to the whole application. While providing an interesting view of the problem and possible solutions, tierless languages have so far not seen significant use by developers. This is perhaps due to the fact that each language starts out from a "clean slate", with no community or ecosystem of their own.

Another, related, strain of research concerns itself with retargeting existing languages towards tierless programming, either by extending the core language or by using existing language features. Languages such as *Eliom* [Radanne et al., 2016], *Conductance* [Fritze, 2014] and *AFAX* [Petricek and Syme, 2007] take the first approach to OCaml, JavaScript and F# respectively, while this thesis concerns itself with applying the second approach, in a mainly Haskell context.

2 Contributions

This thesis describes a set of novel programming techniques for implementing rich, distributed web applications:

- a JavaScript-targeting compilation scheme with accompanying compiler, which significantly outperforms the current state of the art;

- a thorough analysis of common implementation techniques for web-targeting compilers and their impact on the performance of the aforementioned compilation scheme;
- the *Haste.App* programming model for type-safe, distributed web applications;
- a method for implementing self-optimising, high-performance EDSLs for web applications; and
- a solution to the long-standing problem of correctly scoping queries in a monadic database language.

Each technique is accompanied by a proof-of-concept implementation, which is available from the author's website as free software.

Each contribution is described in detail in one chapter of this thesis, with the exception of the *Haste.App* programming model which is covered in two chapters. Each chapter corresponds to a paper, where papers II through V have been published in the peer-reviewed proceedings of various conferences, and papers I and VI are still undergoing preparation for publication.

The remainder of this chapter gives a brief breakdown of each paper, as well as a statement of contributions for each.

2.1 Foundations for Client-Side Haskell Web Applications

Paper I Paper I seeks to corroborate the hypothesis that lazy, functional languages benefit substantially from relying heavily on complex functionality built into the target language, instead of using a straightforward adaptation of a compilation scheme originally targeting a low-level instruction set, when compiled to another high-level language. Most interpreters and JIT compilers for high-level languages contain extensive optimisations for common code paths. Consequently, a related hypothesis of this paper is that hitting said "hot paths" is an important factor in achieving good performance from a compiler targeting such languages.

To this end, paper I develops a compilation scheme and runtime system for a Haskell dialect targeting the JavaScript language rather than a more traditional machine architecture. In the paper, we present a compilation scheme from the STG [Peyton Jones, 1992] intermediate language of the GHC Haskell compiler to JavaScript. In accordance with the aforementioned assumptions, the compilation scheme performs a high-level translation of the STG language into JavaScript. Matching higher-level data and control structures in STG to higher-level structures in the target language, which are

less general than the simple branches and jumps usually produced during code generation, is used to convey higher-level assumptions and invariants directly from the source program to the target language's optimiser.

Unlike when compiling for a native target, and unlike the current state of the art, we do not attempt a lower-level compilation more true to the STG abstract machine. Instead, we rely on the interpreter of the target language being able to recognise high-level code and apply *its own* optimised translation of the target program. A particularly interesting instance of this is explicitly *avoiding* generating tagless code from the STG input, in spite of conventional wisdom, on the assumption that our target language will be able to produce more efficient code from a single branch on a tag than from a more general, considerably heavyweight, function call into a thunk.

Through comparison with GHCJS, the current state of the art in web-targeting compilers for lazy, functional languages, we demonstrate that our compilation scheme produces code which not only runs significantly faster, but is also up to an order of magnitude *smaller*, than code produced by competing compilation schemes. We also perform a survey of relevant optimisation techniques and their impact on the performance of the generated code.

Our work mitigates the problem of lazy, functional programs often being both slow and large. While large code size and significant slowdowns may be acceptable in native binaries, which often have an abundance of processing power and disk space at their disposal, increasing the amount of program code transferred by a large web application by a factor of 10 or even more can be prohibitively expensive. By adopting the compilation scheme proposed in Paper I, implementors of lazy, functional languages can significantly reduce costs to users adopting their languages for use in client-side web applications. The rest of the thesis uses our compilation scheme and the compiler implementing it as a building block for higher-level programming techniques.

Our compilation scheme is implemented in the *Haste* compiler, which is available as free software at https://haste-lang.org.

Paper I is a pre-print version of a paper being prepared for submission to the Journal of Functional Programming (JFP), 2018. It is based on the first chapter of the author's licentiate thesis [Ekblad, 2015].

Paper II This paper details the design and implementation of a novel foreign function interface for a functional language compiling down to some higher-order language. Again, the particular languages used as the source and target languages for our reference implementation are Haskell

and JavaScript respectively.

In the spirit of the compilation scheme described in Paper I – exploit your host environment as much as possible – this interface uses the target language's own lambda abstractions to represent foreign code. Unlike traditional foreign function interfaces, this has the advantage of allowing arbitrary code fragments, not just named functions, to be imported from the target language into the guest language.

Also unlike traditional foreign function interfaces, our interface allows automatic marshalling of arbitrary data, including higher-order functions, between the host and guest languages. Marshalling of non-function data employs a generic traversal, converting source language data structures to structurally equivalent target language data structures, with the exception of "primitive" types – integers, booleans, etc. – which are losslessly converted between their source and target language primitive representations. Function marshalling is slightly more complicated, dynamically allocating new function objects as needed to convert between source and target language calling conventions.

These two key features taken together allows code written using a foreign function interface based on our technique to be significantly more readable, and to avoid a considerable amount of boilerplate. As an example, consider the following code fragment to fetch the current time, written using the standard Haskell foreign function interface:

```
1  data CTimeval = MkCTimeval CLong CLong
2
3  instance Storable CTimeval where
4          sizeOf _ = (sizeOf (undefined :: CLong)) * 2
5          alignment _ = alignment (undefined :: CLong)
6          peek p = do
7                  s   ← peekElemOff (castPtr p) 0
8                  mus ← peekElemOff (castPtr p) 1
9                  return (MkCTimeval s mus)
10         poke p (MkCTimeval s mus) = do
11                 pokeElemOff (castPtr p) 0 s
12                 pokeElemOff (castPtr p) 1 mus
13
14 foreign import stdcall unsafe "time.h gettimeofday"
15    gettimeofday :: Ptr CTimeval → Ptr () → IO CInt
16
17 getCTimeval :: IO CTimeval
18 getCTimeval = with (MkCTimeval 0 0) $ \ptval → do
19   throwErrnoIfMinus1_ "gettimeofday" $ do
20     gettimeofday ptval nullPtr
21   peek ptval
```

Not only is it plagued by considerable amounts of boilerplate code, but

much of that code is low-level enough to be nigh incomprehensible without detailed knowledge of the underlying API. The following code fragment performs the same task, but is written using the reference implementation of our interface:

```
1  data UTCTime = UTCTime {
2      secs  :: Word,
3      usecs :: Word
4    } deriving Generic
5  instance FromAny UTCTime
6
7  getCurrentTime :: IO UTCTime
8  getCurrentTime =
9    host "() ⇒ {var ms = new Date().getTime();\
10               \return {secs:  ms/1000,\
11               \        usecs: (ms % 1000)*1000};}"
```

Contrasting this code fragment with the previous one, we see that not only is the code considerably shorter, but the level of abstraction has also been raised significantly.

In the paper, we describe the design of the interface and give a reference implementation, *fully embedded in the source language*: the target language's dynamic code evaluation facilities are exploited to pass code fragments directly to the its interpreter at run-time, through the source language's built-in, low-level foreign function interface. This property provides significant ease of improvement and experimentation over other approaches, which are normally integrated tightly into the compiler itself. We also demonstrate how the reliance on dynamic code evaluation can be avoided through a minor compiler augmentation. Finally, we give examples to demonstrate that our interface does indeed reduce boilerplate code by a significant amount, and we show through a set of benchmarks, compared against Haskell's standard foreign function interface, that the performance impact of using our interface is in most cases negligible despite its increased expressiveness.

While most languages, Haskell included [Chakravarty, 2003], include perfectly workable interfaces for integrating with other languages, most such interfaces were devised with the intention of interacting with C; the programming lingua franca. Consequently, said interfaces focus on a low-level core API over bytes and pointers. While inconvenient, this detour via a lowest common denominator is often a necessity to bridge the gap between two communicating higher-level languages. However, when compiling to a higher-level language, said higher-level language may instead be used as the lowest common denominator. At this point, the low-level detour is not only inconvenient, but even *increases* the compatibility gap between higher-level languages. Our foreign function interface elegantly solves this

problem while remaining reasonably performant.

This foreign function interface serves as the cornerstone for the reference implementations of the findings presented in papers III through V. Consequently, all implementations can be used with any JavaScript-targeting Haskell compiler, after implementing this interface. As the interface's only compiler-specific components are the stubs for the compiler's native foreign function interface and a small piece of target-language JavaScript for marshalling functions using the source language's particular calling convention, porting the interface to any such compiler is straightforward.

Paper II is based on a paper presented at the Symposium on the Implementation and Application of Functional Languages (IFL), 2015.

Impact The Haste compiler and its accompanying foreign function interface have been used in several functional programming courses at Chalmers University of Technology and one MSc. thesis [Sjösten, 2015]. A BSc. thesis at Chalmers [Block et al., 2016] carried out a more systematic evaluation of the Haste compiler and the Haste.App programming model described in Paper III, in regards to usability and performance, with a generally favourable outcome.

The Haste compiler has seen some use in industrial settings, and the ideas underpinning the FFI described in Paper II have made their way into industry on their own; for instance, the foreign function interface of the Haskell embedding of the *R* language by Tweag I/O [Boespflug, 2015] is explicitly based on Paper II.

The Haste compiler has also garnered some attention in open source circles, so far totalling more than 16 000 downloads from the Hackage package repository alone, with another conservatively estimated 10 000 binary and source code downloads from the project's website and GitHub source repository. Said repository is at the time of writing the 25th most "starred" [1] out of the more than 60 000 Haskell projects on GitHub. Talks about the compiler have been given by independent third parties at industry conferences such as BayHac, CampJS and Strange Loop, in addition to an invited talk by the author at the MLOC.JS web development conference [Ekblad, 2014, Kuhtz, 2014, Miller, 2014, Swenson-Healey and Cooper, 2014].

2.2 Type-Safe, Distributed Web Applications

Paper III This paper presents a tierless programming model for implementing rich, client-server web applications using only standard Haskell,

[1] A measure of popularity, akin to a Facebook "like".

and the *Haste.App* library implementing it. Web applications, traditionally implemented as two or more independent programs communicating using some ad hoc communication protocol, are in this model written as a single Haskell program. The client and server parts of the application are separated by the type system, with client-side code residing in a *Client* monad and server-side code in a corresponding *Server* monad. The computation is driven by the client side, with the server lying dormant until a client makes a remote procedure call to some function in the Server monad; the server may not call back into the the client on its own volition. This restriction helps keep the program flow clear and explicit.

A key component of this programming model is that programs are compiled not once, but *twice* – once to produce a server binary, and once to produce client-side JavaScript code. The supporting *Haste.App* library splits the application in two parts, ensuring that code executing in the Client monad ends up on the client, and that code executing in the Server monad ends up on the server. A third monad is used as a staging area, where server-side functions are "imported" onto the client, to provide the glue between the client and the server. This code is executed on the client as well as the server, but performs a slightly different task depending on where it executes: on the client, it connects source-language identifiers to their corresponding server-side RPC endpoints, whereas on the server, it builds a lookup table to map client RPC calls to their server-side implementations. Pure code – that is, effect-free code callable from any monad – is duplicated, and ends up on both client and server.

This automatic splitting is achieved by compile-time introspection. When the library detects that the program is being compiled for the client, it filters out any server-side code, replacing all calls to such functions with stubs taking care of the network communication and synchronisation necessary to make a remote call. Similarly, when the library detects that the program is being compiled for the server, all client-side code is filtered out and the program's entry point is replaced by an event loop, dispatching server-side functions to fulfil remote calls as they come in from clients.

As previously mentioned, there exists a wealth of previous work on tierless web applications. What sets our programming model apart is its reliance only on existing tools for the Haskell language: it can be implemented entirely as a library, without the need for new languages or compiler modifications. In addition to lending itself well to agile experimentation, this has the benefit of letting it ride on the coattails of the Haskell community and infrastructure, as developers can combine it with their favourite Haskell tools, libraries and idioms right out of the box. A key insight in enabling this is the use of multiple compilers to produce different binaries from

a single program, with the supporting Haste.App library controlling the placement of code fragments.

Paper III is based on a paper presented at the Haskell Symposium, 2014, coauthored with Koen Claessen.

Paper IV While the programming model presented in paper III works well for single-server applications, its model of a web application is overly simplistic. Commonly, an application does not consist of a client and a single server, but of a client and *multiple* servers, which are often heterogeneous in nature. Computational resources, databases, ad servers, etc. are all common occurrences in present-day web applications.

In paper IV, we generalise the results from paper III to cover an arbitrary number of interconnected servers. We keep the basic premise of using two compilers to produce client and server executables, but generalise it to allow any number of different binaries to be produced by any number of compatible compilers. We also keep the idea of determining code placement based on the type of an expression, but again generalise this concept from two designated client and server monads to any number of different nodes. Communication between nodes is still handled like "normal", type-safe, monadic function calls, with the underlying network code being generated by the supporting library.

We also lift the requirement that nodes must be monadic, allowing, for instance, applicative and arrow nodes. This is partially to better support *exotically typed* nodes – nodes on which calculations are performed in another type universe than on the calling client. Tight integration with exotically typed nodes reduces the amount of boilerplate code required to directly connect nodes written in some embedded, domain-specific language to the network, as even EDSLs with very different programming models – SQL queries or GPU kernels, for instance – can be called by clients without having to deal with type conversions or the details of how to compile, load and execute the embedded programs.

Nodes are connected in a directed graph, where the client node – the one executing in the user's browser – is transitively connected to all other nodes. The client still drives the computation, but server nodes may now themselves be clients of *other* server nodes as well. A node is connected to the network by instantiating a type class, describing how it may be reached by other nodes, which other nodes may communicate with it, how its type universe maps to that of any calling client, and whether the node needs any particular initialisation. For nodes which are not exotically typed, most of these properties are optional, defaulting to values appropriate for most nodes written in "normal" Haskell.

We further generalise the concept of nodes to cover virtual servers as well, and demonstrate how this lets us model fine-grained sandboxing of untrusted JavaScript code as simple RPC calls. Web applications often load third party code from external sources at runtime, which makes them vulnerable not only to being compromised themselves, but to security breaches on any network or remote host from which code is loaded. Our sandboxes-as-servers method exploits browsers' built-in sandboxing mechanisms, which are normally too coarse-grained and cumbersome to see widespread use, to implement convenient, fine-grained sandboxing.

These generalisations sets Haste.App further apart from other tierless web programming models, which generally only support a fixed set of nodes under a fixed set of programming paradigms.

Paper IV is based on a paper presented at the Haskell Symposium, 2017.

Impact Like the Haste compiler, a systematic evaluation of the initial Haste.App programming model, described in paper III, was carried out at Chalmers University of Technology, with favourable results both regarding performance and usability. While industry interests have been rather more reluctant to embrace Haste.App than the Haste compiler, it has been used with positive results to implement research and teaching software at Chalmers as well as at other institutions [Kahl, 2016].

2.3 Domain-Specific Problem Solving through EDSLs

Paper V This paper describes a method for integrating high-performance, low-level computational kernels into Haskell web applications, and an accompanying proof of concept EDSL.

While concise, high-level and readable code is often preferable to highly performant code for most applications – developer time being significantly more expensive than processing time – some applications may benefit from having both. For instance, online games, signal processing and cryptographic applications may all have performance-critical bottlenecks where the performance penalty imposed by higher-level languages is unacceptable, while large parts of the application – such as user interfaces and data storage – may not be very performance sensitive at all. In a traditional setting, such situations are often resolved by writing the performance-critical parts of an application from C and integrating them into the higher-level application using some foreign function interface, but in a web application there is no such recourse.

Paper V provides a solution to this problem, in the form of the *Aplite* low-level EDSL geared towards computationally heavy tasks. We build on the results from the DSP-targeting Feldspar language [Axelsson et al.,

2010], but adapt the methods to a web context, exploring a multi-backend compilation scheme targeting both *ASM.js* – a subset of JavaScript designed to be highly optimisable – and plain but efficient JavaScript.

We demonstrate a method to seamlessly integrate Aplite kernels into the Haskell host program, making them indistinguishable from "normal" Haskell functions, even though Aplite kernels have a completely different type universe from its Haskell host. Kernels with host-observable side-effects can be imported as plain Haskell functions in the IO monad, whereas kernels with only local side-effects may be imported either as pure or monadic functions, depending on which type best suits the programmer. This is accomplished by leveraging the dynamic code loading capabilities of the foreign function interface described in paper II and judicious application of type-level functions. Being first-class objects, Aplite programs represent an application of multi-stage programming, allowing the host program to specialise Aplite programs to its runtime environment as well as user input and other parameters.

Recognising that different applications may have wildly different performance characteristics depending on the combination of backend, browser environment and user input they are presented with, Aplite supports recompiling existing kernels with different parameters. More interestingly, we demonstrate a method whereby a kernel is *automatically* profiled with a series of different compilation parameters, and the most efficient implementation selected by any subsequent calls to the kernel.

We thoroughly benchmark our reference implementation against web-targeting Haskell code as well as hand-rolled JavaScript, and demonstrate that our language outperforms both on all investigated benchmarks. In particular, we demonstrate that backend selection must be informed by both the performance characteristics of the kernel in question and the current browser environment, giving solid evidence for the efficacy of our multi-backend compilation scheme and profile-guided backend selection.

Paper V is based on a paper presented at the Haskell Symposium, 2016.

Paper VI This paper presents a simple but effective solution to the long-standing problem of ensuring the well-scopedness of a monadic formulation of relational database queries, together with a simplified version of *Selda*, the first relational database EDSL to support both a well-scoped monadic interface and fully general[2] inner queries.

While there exists an ample body of previous work in EDSLs for integrating with relational databases, so far none has managed to ensure that queries are well-scoped in the presence of fully general inner queries, while

[2] As opposed to static SELECT statements over fixed tables.

maintaining a monadic interface. Monadic interfaces are useful for Haskell EDSLs, as they provide a familiar and well understood interface to user and developer alike, with good support from standard and third-party libraries.

Consider the following monadic pseudocode query, intended to associate each person with their home city, but only if said city is located in Sweden.

```
1  addresses = do
2    (name :*: addr) ← select persons
3    city ← leftJoin (\city → addr .== city) $ do
4      (city :*: country) ← select cities
5      restrict (country .== "Sweden")
6      return city
7    return (name :*: city)
```

A straightforward translation into SQL presents us with the following query.

```
1  SELECT personName, cityName
2    FROM persons
3    LEFT JOIN (
4      SELECT cityName
5      FROM cities
6      WHERE cities.country = "Sweden"
7    )
8    ON persons.address = cityName
```

While this query is not problematic per se, the fact that the *name* and *addr* identifiers are in scope *inside the body of the left join* is a cause for concern. In fact, this enables the creation of decidedly nonsensical queries, as in the following modification of the above example.

```
1  illScopedAddresses = do
2    (name :*: addr) ← select persons
3    city ← leftJoin (\city → addr .== city) $ do
4      (city :*: country) ← select cities
5      restrict (country .== "Sweden")
6      restrict (city .== addr)
7      return city
8    return (name :*: city)
```

Here, the body of the join refers directly to the *addr* identifier, even though no table referenced by the inner query has any such field; the query is *ill-scoped*. Clearly, any type-safe relational database EDSL must disallow such nonsensical queries.

This paper presents a simple way to ensure the well-scopedness of inner queries based on type-level functions over phantom types. Like the standard Haskell *ST* monad [Launchbury and Peyton Jones, 1994], Selda solves the problem by parameterising its query monad over a phantom

type denoting its scope. Expressions in the monad are then also augmented with a scope parameter, ensuring that computations can only operate on expressions within its own scope.

Unlike the ST monad, which only allows pure values to be returned from stateful computations, inner queries in a database EDSL must be able to return *EDSL expressions* to the outside world. As this is not possible using the method employed by the ST monad, its type parameter being existentially quantified, we instead view the scope parameter as a *scope counter*. The outermost query has a scope equivalent to zero, and each nesting of an inner query increments the scope counter by one. Expressions returned from an inner query have their scope counter decremented by one, to allow the outer query to operate on them, but crucially, expressions are *not* able to migrate *inward*, solving the scoping problem introduced previously.

We also discuss the similarities and differences between general inner queries and inner aggregate queries, show how a similar problem arises when compiling aggregated queries to SQL, and demonstrate how the scope counter solution may be applied to the aggregate compilation problem. Finally, we present a simplified version of the Selda API, demonstrating how the scope counter solution can be incorporated in the language to provide a simple, monadic interface that supports fully general inner queries while ensuring their well-scopedness.

Paper VI is based on a paper presented at the Symposium on Trends in Functional Programming, 2017, and being prepared for submission to the 2018 Haskell Symposium.

Impact While not as popular as the Haste compiler, the Selda library has in its nine months of existence become the fourth most downloaded database EDSL on the Hackage package repository, as well as one of the 200 most popular Haskell repositories on GitHub out of more than 60 000. As paper VI is as yet unpublished academic interest has been scarce, but the community surrounding the library includes several industrial users.

Aplite, on the other hand, has so far not garnered any significant industrial or open source interest. With the rapid advance of WebAssembly [Eich, 2015] rendering ASM.js largely obsolete, this is not expected to change without significant retargeting and repackaging efforts.

3 Background

3.1 Functional Programming

Functional programming is a discipline of software development which views programs as functions from inputs to outputs, built from smaller functions which are in turn built from *even smaller* functions, and so on. Unlike the more familiar functions encountered in high school, a functional program does not restrict itself to operations over, say, real numbers. Mouse movements, real-time audio streams and even other programs are all examples of possible inputs, while outputs may include the sending of messages over a network, pixels displayed on a screen, or haptic feedback through a game controller.

Functional programs are declarative: the programmer describes the relations between the application's states, focusing on *what* the application is supposed to be doing. In contrast, imperative programs consist mainly of code describing *how* the program is intended to achieve its goal.

Higher-order functions The *functional* in functional programming is perhaps most apparent in its treatment of functions as *first-class objects*: just like integers or floating point numbers, functions are just another type of data to be created, passed around and bound to identifiers – or not, as the programmer chooses. This enables powerful forms of decoupling and abstraction, where functions may leave "holes" of undefined behaviour, to be filled in by its caller.

For instance, consider the following implementation of the `merge` function, which merges two lists which are sorted in ascending order, into a single list which is also sorted in ascending order:

```
1  merge :: Ord a ⇒ [a] → [a] → [a]
2  merge (x:xs) (y:ys) =
3    if x < y then x:merge xs (y:ys)
4             else y:merge (x:xs) ys
5  merge _ [] ys = ys
6  merge _ xs [] = xs
```

While this function is certainly useful, perhaps to display two separate, ordered data sources to a user as a single table, it is not very flexible: what if we sometimes need to merge two lists sorted in *descending* order? We could, of course, implement *two* functions – `mergeAscending` and `mergeDescending` – but by the *DRY*[3] principle [Hunt and Thomas, 2000], we really shouldn't.

A better solution would be to parameterise the function's behaviour over the way in which we want to sort the elements:

[3] Don't Repeat Yourself

```
1  merge2 :: Ord a ⇒ Bool → [a] → [a] → [a]
2  merge2 ascending (x:xs) (y:ys) =
3    if ascending
4      then if x < y then x:merge2 ascending xs (y:ys)
5                    else y:merge2 ascending (x:xs) ys
6      else if x > y then x:merge2 ascending xs (y:ys)
7                    else y:merge2 ascending (x:xs) ys
8  ...
```

While this solution certainly contains less repetition than writing two separate functions, it is still not ideal. We don't entirely get rid of repetition and, most importantly, *we can only support merging behaviours that the original implementer of `merge2` could foresee!* This is a real problem when working with data that does not necessarily have a single, canonical total ordering but which we still may want to sort somehow: tuples of numbers sorted by some mathematical property, or cartoon ponies sorted by their suitability for some given task, for instance.

Instead, in a functional program we would parameterise the merge function, not over a flag to choose one of several hard-coded comparison behaviours, but *over a function describing the comparison itself*:

```
1  mergeBy :: Ord a ⇒ (a → a → Bool) → [a] → [a] → [a]
2  mergeBy goesBefore (x:xs) (y:ys) =
3    if x `goesBefore` y then x:mergeBy goesBefore xs (y:ys)
4                        else y:mergeBy goesBefore (x:xs) ys
5  ...
```

By simply leaving the choice of the comparison function up to the caller, we gain several important advantages: we no longer need to implement different behaviours depending on some user-supplied flag, we get rid of the repetition inherent in doing so, and – most importantly – we separate the task of merging two lists from the task of comparing two elements.

Functions that accept other functions as inputs are known as *higher-order functions*, and are the bread and butter of functional programming, providing programmers with a natural means to modularise their programs, with any desired level of granularity.

Function composition The treatment of functions as first-class objects enables us to write higher-order functions to *compose* functions in various, often highly generic, ways. As a example, Haskell provides a standard function composition operator to compose two functions `f` and `g` by creating a new function which first applies `g` to its argument `x`, and then applies `f` to the result of `g`:

```
1  (.) :: (b → c) → (a → b) → (a → c)
2  f . g = \x → f (g x)
```

Almost trivial in its definition, standard function composition is surprisingly useful, allowing many complex functions to be expressed as a *pipeline* of smaller, less complex, functions.

As an example, consider the following function:

```
1  toSet :: (Ord a, Eq a) ⇒ [a] → [a]
2  toSet = map head . group . sort
```

This function performs the nontrivial task of turning an unsorted list of possibly duplicate elements, into an ordered set: a list which is sorted and guaranteed to contain no duplicate elements. Instead of tackling the whole task at once, we construct the solution as a pipeline: first we `sort` the input list, then we `group` all adjacent elements that are equal to each other into sub-lists, and finally we extract the `head` – or first element – of each such sublist.

Compared to a more monolithic solution, it is easy to convince oneself that this function is correct: if two or more elements are equal, then they must all be adjacent to one another after sorting; if two or more equal elements are adjacent to each other, they must all end up in the same sublist after grouping; ergo, extracting the first element of each such sublist trivially gives us a single representative for each group of equal elements.

Note how this manner of programming plays into the aforementioned theme of modularity: the `toSet` function is a simple composition of prebuilt, generic functions, with no conditionals or logic of its own save for the choice of functions used in its implementation and the order in which they are composed. This is a great boon to modularity and code reusability, which in turn brings substantial benefits for programmer productivity [Hughes, 1989].

Higher-order functions as OOP design patterns Readers familiar with object-oriented design patterns [Gamma et al., 1993] may notice certain similarities between higher-order functions, function composition, and several design patterns: the *command*, *visitor*, *observer*, *strategy* and *dependency injection* patterns directly correspond to different specialised uses of higher-order functions, while patterns such as *bridge*, *facade*, *adapter*, *builder* and *proxy* can be easily implemented using function composition.

Pure functional programming While functional programming in general can be beneficial to programmers, the Haskell programming language used

throughout this thesis takes the functional programming paradigm one step further, in its adoption of *pure functional programming*.

In a purely functional programming language, all functions are functions in the *mathematical* sense: the output of a function depends solely on its inputs. That is to say, a function may not give a different result depending on the number of times it has been called, the current date, or any other information that may vary depending on the program's circumstances. A pure function must also not perform any *effects* – changing the state of the program or its environment – aside from computing its value. Consequently, data in a purely functional program is *immutable*.

Immutability brings a host of benefits for programmers: it is easier to reason about programs without ever-changing program state, and bugs are easier to prevent, diagnose and rectify in a program with fewer "moving parts". The advantages of immutability are widely recognised, and its application is recommended even for languages with relatively poor built-in support for enforcing immutability [Bloch, 2008].

Above all, pure computations are highly composable, as they do not make any assumptions about the context from which they are called, only relying on their inputs. This makes the dependencies of program units explicit, reducing the cognitive burden on the programmer when assessing whether a particular modification will affect another part of the program. This property is particularly important for web applications, which are mainly event-driven and continuation based, making their flow of execution impossible to predict.

Enforcing purity through types While immutability by convention is indeed an important step up from a programming style based on gratuitous mutation, it still leaves many things to be desired. It is not always trivial to convince oneself that a piece of code written in, say, C# or JavaScript is indeed pure: as purity is not tracked across compilation units – or even classes or methods – mutation may hide in the murkiest code depths, many layers of indirection removed from the call sites where we would like to verify its immutability. The allure of sacrificing compositionality by turning to mutation as a quick and dirty fix, with only programmer discipline to keep it in check, only serves to exacerbate this problem.

To fully reap the benefits of purity, Haskell encodes the effects a function may perform in its type, encapsulating computations with side effects in a type of their own called `IO`. If an integer has the type `Int`, then a computation that produces an integer has the type `IO Int`; and if a function which accepts a string as its input and returns an integer has the type `String -> Int`, then a function from strings to integers which *also* may perform some effect

has the type `String -> IO Int`. This ensures that purity is tracked and enforced throughout programs: if we have a function `plusOne :: Int -> Int`, applying it to a computation of type `IO Int` rather than a plain `Int` will cause a type error during compilation.

At first glance, this would seem to preclude the use of pure functions to manipulate user input, as it is plainly impossible for a function which reads user input – thereby depending on the state of the world – to be pure. Fortunately, the `IO` type is an example of a *monad* [Wadler, 1995]: an abstraction which allows values contained in another type to be manipulated by pure functions, the resulting new value safely re-encapsulated within the containing type again. Thus, in a Haskell program, impure code may depend on pure functions, but pure code may *not* depend on impure functions.

3.2 Embedded Domain-Specific Languages

Embedded, domain-specific languages, or *EDSLs*, are a software design pattern in which different problem domains of an application are addressed using different programming styles, effectively creating a set of task-specific sub-languages *inside* the general-purpose host language. It is related to the object-oriented *interpreter*, or *little languages* design pattern, where an application outsources some problem domain to an *external* domain-specific language, which the main application then interprets [Bentley, 1986].

The main difference between the two design patterns lies in the *embedded* part: where an external domain-specific language has the freedom to implement any conceivable language, an EDSL instead piggy-backs on the capabilities of the host language. While an EDSL does not enjoy the same degree of freedom as its non-embedded kin, it also does not incur the same implementation and usage overheads. At the cost of being bound by the host language's restrictions, the EDSL gains the use of the host language's parser, type system, runtime system, tooling, and so on.

EDSLs have a strong tradition in the functional programming community [Hudak, 1996], but has also seen significant adoption in other, more mainstream, communities, with high-profile projects such as Tensorflow [Abadi et al., 2016] and RSpec [Chelimsky et al., 2010] being built largely on this idiom, in Python and Ruby respectively.

It can sometimes be hard to make the distinction between an EDSL and a particularly focused library. Ultimately, this is often a matter of branding, and the level of cohesion between the components making up the EDSL/library. In this thesis, we take the view that an EDSL is a library where the components are intended to be used almost exclusively together in a cohesive manner to solve problems in some particular domain, as

opposed to being thinly sprinkled across a code base largely consisting of "other" host language code.

EDSLs and types While the EDSL design pattern can be powerful in any language, the dynamic type system and reliance on convention offered by common implementation languages such as Ruby and Python can sometimes, perhaps counter-intuitively, *restrict* the applicability of EDSLs.

As a motivating example, let us look at the Haskell *STM* EDSL, which allows the programmer to implement communication in a concurrent program as a set of *transactions* over shared mutable variables. STM works on the principle that, just like with relational database transactions, most concurrent data accesses do not interfere with each other, and the overhead of locking shared data is thus often unnecessary. Additionally, handling multiple locks at once is a subtle and error-prone business, and is best avoided whenever possible.

Instead, STM programs are separated into transactions, where each transaction reads and writes shared references with impunity, only committing the result of *all* writes at the end of the transaction. If any shared reference accessed by the transaction was modified at some point during the transaction, the result is not committed but the whole computation is instead *retried* until it succeeds.

It is easy to see that this programming model places some heavy restrictions on the programmer: any code placed within a transaction must be free from effects, as the transaction may be retried any number of times before finally being committed. If any piece of the transaction causes, say, an intercontinental missile to be launched, high contention over shared resources may cause us to retry the transaction a significant number of times, thereby exhausting our stockpile of missiles even though we only intended to fire one!

It is equally easy to see that this EDSL would be depressingly unsafe if implemented in a language where purity is not enforced by the compiler. As pointed out in Sect. 3.1, manually ensuring that any piece of code is indeed pure is a non-trivial task – one that we may want to avoid altogether if the correctness of our application depends on it.

In Haskell, by contrast, this property can be almost trivially guaranteed by leveraging the type system. The monad concept used to model effectful computations in Haskell, turns out to be a flexible, general abstraction for implementing EDSLs, providing a greater level of cohesion and isolating EDSLs from each other as well as from impure host language code [Hudak, 1996]. The STM EDSL uses this to great effect by giving all transactions the type `STM a`, where `a` can be any type. By not including an operation in the

language to turn an IO computation into an STM computation, the risk of effectful computations being executed more than once due to transaction retries is completely eliminated.

The power of Haskell's type system allows us to restrict the behaviours of embedded programs even further, for instance by disallowing pure computations over types not well suited to the problem domain [Axelsson et al., 2010, Bracker and Gill, 2014, Svenningsson and Svensson, 2013] or by enforcing custom scoping rules [Launchbury and Peyton Jones, 1994].

In essence, when it comes to EDSLs, it is sometimes the case that "less is more", and that by restricting embedded languages to their target problem domain – and that domain *only* – we open up additional opportunities for safe and efficient problem solving.

4 Bibliography

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 169–178. IEEE, 2010.

J. Bentley. Programming pearls: little languages. *Communications of the ACM*, 29(8):711–721, 1986.

J. Bloch. *Effective Java*. Addison-Wesley, 2008.

B. Block, J. Gustafsson, M. Milakovic, M. Nilsen, and A. Samuelsson. Evaluating Haste.App: Haskell in a web setting. Effects of using a seamless, linear, client-centric programming model, 2016.

M. Boespflug. Haskellr. https://tweag.github.io/HaskellR/, 2015.

J. Bracker and A. Gill. Sunroof: A monadic DSL for generating JavaScript. In M. Flatt and H.-F. Guo, editors, *Practical Aspects of Declarative Languages*, volume 8324 of *Lecture Notes in Computer Science*, pages 65–80. Springer International Publishing, 2014. doi: 10.1007/978-3-319-04132-2_5.

M. M. Chakravarty. *The Haskell Foreign Function Interface 1.0: An Addendum to the Haskell 98 Report*. 2003.

D. Chelimsky, D. Astels, B. Helmkamp, D. North, Z. Dennis, and A. Hellesoy. The RSpec Book: Behaviour Driven Development with RSpec. *Cucumber, and Friends, Pragmatic Bookshelf*, 2010.

A. Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 122–133, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. doi: 10.1145/1806596.1806612.

E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-74791-8. doi: 10.1007/978-3-540-74792-5_12.

B. Eich. From ASM.js to WebAssembly. https://brendaneich.com/2015/06/from-asm-js-to-webassembly/, 2015.

A. Ekblad. Hastily paving the way for diversity. http://www.ustream.tv/recorded/43804744, 2014.

A. Ekblad. A distributed haskell for the modern web. 2015. Also available from https://ekblad.cc/pubs/haste-licentiate.pdf.

A. Fritze. The Conductance application server. http://conductance.io, 2014.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*, pages 406–431. Springer, 1993.

P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.

J. Hughes. Why functional programming matters. *The computer journal*, 32 (2):98–107, 1989.

A. Hunt and D. Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional, 2000.

W. Kahl. CalcCheck: A Proof-Checker for Gries and Schneider's "Logical Approach to Discrete Math". http://calccheck.mcmaster.ca/, 2016.

L. Kuhtz. Haste: Front end web development with haskell. https://www.youtube.com/watch?v=Arot_cDmQHI#t=220, 2014.

J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 24–35, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X. doi: 10.1145/178243.178246.

K. Miller. Make haste: Fast track fo functional thinking. https://www.youtube.com/watch?v=o3JMxnnTZ64, 2014.

D. Namiot and M. Sneps-Sneppe. On micro-services architecture. *International Journal of Open Information Technologies*, 2(9):24–27, 2014.

T. Petricek and D. Syme. AFAX: Rich client/server web applications in F#. 2007.

S. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2: 127–202, 1992. ISSN 1469-7653. doi: 10.1017/S0956796800000319.

G. Radanne, J. Vouillon, and V. Balat. Eliom: A core ML language for tierless web programming. In *Asian Symposium on Programming Languages and Systems*, pages 377–397. Springer, 2016.

D. Rajchenbach-Teller. Opa: Language support for a sane, safe and secure web. *Proceedings of the OWASP AppSec Research*, 2010.

A. Sjösten. SWAP-IFC: Secure Web Applications with Information Flow Control. 2015.

J. D. Svenningsson and B. J. Svensson. Simple and compositional reification of monadic embedded languages. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 299–304, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500611.

E. Swenson-Healey and J. Cooper. Haste: Full-stack haskell for non-phd candidates. https://www.youtube.com/watch?v=3v03NFcyvzc, 2014.

P. Wadler. Monads for functional programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer, 1995.