School of Computer Science and Statistics

Modular Language Stacking

Ernests Kuzņecovs

April 21, 2022

A dissertation submitted in partial fulfilment of the requirements for the degree of Masters of Computer Science
Declaration

I hereby declare that this dissertation is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at http://www.tcd.ie/calendar.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at http://tcd-ie.libguides.com/plagiarism/ready-steady-write.

Signed: ___________________________          Date: ___________________________
Abstract

Oftentimes software systems have trouble decoupling the different domains that they’re built up from, causing software teams to be responsible for multiple layers of the software stack at once.

The Modular Language Stacking paradigm is described and proposed as a solution to domain contamination. The benefits of large scale Modular Language Stacking are hypothesised, and seem to have far reaching positive consequences.

The extremely high level Functional Reactive Programming paradigm is used as a vehicle to understand how a totally pure domain specific language may be developed and how should the description of Modular Language Stacking be refined. From this, the intuition is gathered that extensible, deeply embedded domain specific languages are required. Furthermore the extensible deep EDSLs are to be implemented in terms of other extensible deep EDSLs, achieved through compilation processes on the abstract syntax trees of the EDSLs.

Haskell literature on EDSLs is surveyed to gauge how well we are situated towards actualising Modular Language Stacking. The rich literature orients us and gives us a clear perspective for the next steps.
Acknowledgements

Special thanks to my supervisor, Glenn Strong, for trusting and allowing me to pursue my budding interest in this topic, the natural engagement of the topic has washed me wave after wave, evolving my perspective and accumulating knowledge and understanding.

Thank you to my second reader, Stephen Barrett, for engaging with my demonstration and giving critical feedback on conveying my explorations in reader-motivating manner.

Thank you to HubSpot for taking me on as an intern for two summers on different roles which has exposed me to industry practices, and in turn planting and incubating the vision which is responsible for this work.

Thank you to Conal Elliott for answering my questions on the Functional Programming Slack and discussing with me the importance of pure programming language abstractions, the discussion lead to a major turning point in my perspective and the research.
## Contents

1 Introduction ............................................. 1
   1.1 Contributions ........................................... 1

2 Background and Problem ................................. 3
   2.1 Contaminated Domains ................................. 3
   2.2 Suitable Domain Interface: Functional Reactive Programming .......................... 5
      2.2.1 Functional Reactive Programming .................... 5
      2.2.2 FRP vs. Imperative Programming .................... 6

3 Modular Language Stacking ............................. 10
   3.1 Modular Languages ...................................... 10
   3.2 Language Stacking ...................................... 11
   3.3 Balancing Compilation and Abstract Language Features ........................... 13
   3.4 Large Scale MLS ....................................... 14
      3.4.1 Organisation Structure ........................... 14
      3.4.2 Swapping out and Extending Implementation ................. 14
      3.4.3 Specialised Organisations .......................... 15
      3.4.4 Open Access Application Interface .................... 15
      3.4.5 Data Usage Transparency & End-User Control ............... 16
      3.4.6 Responsible Data Handling .......................... 16
      3.4.7 Program Stack Awareness & Security through Minimalism ........ 16

4 Toward Actualising Modular Language Stacking ......... 17
   4.1 Embedded Domain Specific Languages in Haskell .......................... 17
      4.1.1 Shallow vs. Deep Embeddings ........................ 18
      4.1.2 FRP Embeddings .................................... 19
   4.2 EDSL Techniques ....................................... 21
      4.2.1 Modular Abstract Syntax Trees: Compositional Data Types ............ 21
      4.2.2 Modular Tree Processing: Modular Tree Automata ............... 26
1 Introduction

Software development is an extrememly new practice relative to the history of humans, and it is almost impossible to imagine a future where the relationship between humans and software will cease to exist.

It is therefore important to consciously decide and steer the direction of software practices to the best of our imagination, especially since software systems, in quite a literal sense, are a creation entirely of our own.

The Modular Language Stacking paradigm is described as the next evolutionary step for the human-software relationship, with the far reaching consequences of empowering both, the software developer, and the software consumer.

1.1 Contributions

The contributions are as follows:

- Identify and describe how contemporary web app software contaminate their domains, section 2.1.
- Describe how Functional Reactive Programming is suitable as an abstraction for web apps, section 2.2.
- Propose and describe Modular Language Stacking as the paradigm to support pure Functional Reactive Programming for general application, chapter 3.
- Extrapolate the benefits of large scale Modular Language Stacking, section 3.4.
- Make distinct the different types of embedded domains specific languages that can be defined in Haskell, and identify which types are most relevant for Modular Language Stacking, subsection 4.1.1.
- Identify the limitations of current Functional Reactive Programming implementations through the lens of Modular Language Stacking, subsection 4.1.2.
• Survey and describe Haskell EDSL techniques that can serve as scaffolding for Modular Language Stacking, section 4.2.
2 Background and Problem

2.1 Contaminated Domains

In the context of the industry handling customer data, it is common to use the tried and tested Relational Database along with a general purpose programming language to interface with the database. Pseudo Haskell listings 2.1, 2.2, approximate what the flow of interacting with the Relation Database might look like.

In 2.1, computing some value from 2 queries, this can be thought of a job that is scheduled to run periodically after certain time intervals, i.e a cron job. In 2.2, is a mock of listening for requests and responding.

The main observations are that:

- Algebraic Datatypes should allow for expressing the application domain to our liking, though the datatypes must be derived from the layout of the schema since it is a major source of the data in the application.

While the schema is devised with the application domain in mind, the schema is also devised to adhere to certain performance and layout characteristics for ease of relational queries and extensibility i.e normal forms.

The programmer is either left manually translating to and from the database layout and ideal application domain layout, or the programmer is left contaminating their application layer functions with database layer details.

Listing 2.1: Cron job

```haskell
scheduledJob :: IO ()
scheduledJob = do
  val1 <- dbQuery1
  val2 <- dbQuery2
  let val3 = operate val1 val2
  dbStoreVal val3
  return ()
```
Listing 2.2: Web server

```haskell
listen :: IO ()
listen = runServer handleRequest

handleRequest :: Request -> IO ()
handleRequest =
  \req -> case req of
    ... -> respond1 req
    ... -> respond2 req

respond1 :: Request -> IO ()
respond1 r = do
  dbStore1 r
  q <- dbQuery1 r
  let res = mkResponse r q
  serverRespond res

respond2 :: Request -> IO ()
respond2 r = do
  q <- dbStore2 r
  res = mkResponse2 r q
  serverRespond res
```

- Programming model is mixing both implementation and application layer logic. The code is explicitly calling database queries in a procedural manner. An alternative interpretation of the application layer logic is not possible.

- Concurrency has to be explicit, a manual effort is needed to by the programmer to identify and make beneficial concurrent database operations.

Highlighting why such properties are often undesirable:

- A function which explicitly calls databases may require mocking of layers lower in the stack. Keeping the application logic isolated from implementation details allow for direct testing on the isolated logic, and the implementation can be mocked for the higher layer instead.

- The application layer programmer, needs to be concerned with details such as data layout and performance, rather than being concerned with just the high level application itself.
  - Making application layer changes might trigger changes to the implementation layer. This makes approximating task completion time less predictable.
  - High coupling of layers removes the possibility for specialised teams dedicated to just application logic and teams dedicated to just implementation logic.
The programmer requiring to spend time on both layers, makes specifying more of the application application layer slower, incentivising splitting up the layer into parts that other individuals can work on, resulting in a more fragmented surface layer. This is discussed further in subsection 3.4.1.

- Sequential-first programming is an indirect modelling of the domain when the nature of the domain is concurrent, which is often the case for the web-based software systems services to multiple users at once.

2.2 Suitable Domain Interface: Functional Reactive Programming

Functional Reactive Programming (FRP) [1] is proposed as direction towards remediating the problems highlighted in 2.1.

FRP can be thought of as a programming paradigm which is:

- Inherently concurrent.

- A high level framework to define the relations between data types without needing to specify anything about the implementation.

2.2.1 Functional Reactive Programming

This subsection is to introduce and to get a feel for the FRP paradigm.

The core constructs in FRP are the "Event", "Behavior", and the combinators between them. An "Event a" represents values of the datatype "a" that occur in moments of time. A canonical example of something that is modelled by Event are the clicks of a mouse. "Behavior a" represents a time-varying value of type "a", unlike "Event a", a value of "a" is always present and is can be "sampled" at any time to yield a value of "a". Examples of something that is modelled by Behavior are the mouse position and pixel colour on the screen.

In 2.3 a few of the combinators are shown off:

- "sampleWith" uses an "Event a" to trigger a sampling from a "Behavior b", the result being an "Event b", where the "Event b" fires whenever the "Event a" fires, but will have a value of type "b" which is sampled out of the Behavior at that moment in time. In the example sampleWith is used to extract the mouse position whenever a left-click happens.

- "filterEvent" takes an "Event a" and a predicate on "a", and returns a new "Event a"
which fire the values of the original "Event a", but only the ones that evaluate to true according to the predicate. In the example, filterEvent is used to only listen to left-clicks.

- "accumulate" uses an "Event (a -> a)" to update the value of an "a". Every time the Event fires, the function that is within the Event is applied onto the previous "a", and the result becomes the current "a", represented by the "Behaviour a".

- Behavior and Event are instances of Functor and Applicative. In the example, mapEvent is used to transform the "Event MousePosition" into an "Event ([MousePosition] -> [MousePosition])".

The example collects the mouse positions where the mouse has performed a left click, and transforms the history of left clicks into a heatmap of the clicks.

Every time the Event "leftClickedPoint" produces a new value, the function (:) is curried onto the new value inside the event of "leftClickedPoint", leaving us with the signature "[a] -> [a]" which is immediately then applied to the current value "leftClickHistory", updating it to the new, resulting "[a]". The "accumulate" combinator can be thought of as having a recursive nature, where the resulting Behavior is updating its values based on old versions of itself.

2.2.2 FRP vs. Imperative Programming

An example of an FRP webserver would be listing 2.4.

In the code, we are not explicitly instructing what needs to be persisted. Our only interface to the data are through the Events and Behaviours. So rather than basing our datatypes on the database schema, we are creating datatypes as the application domain requires. We can construct arbitrary intermediate Behaviors and Events from other Behavior and Events.

Concurrency is implicit, we are expressing the semantic relationship between the data directly.

The code is not directly specifying how the implementation will persist the data, compute intermediary data-structures, prioritise resource allocation, etc. It can be understood that the language is fitting the application and only the application, preventing domain contamination. Methods for attaching an interpretation or implementation to the FRP network specification is discussed in 3.2.

We are assuming an interface to the outside world, i.e where the "Event RawRequest" come from, and how were sending Responses. These features can be thought as specific to the application domain itself, in this case, a web server. It can be understood that these specialised interfaces to the outside world are native to the domain language. This aspect
Listing 2.3: FRP example

data MouseClick = LeftClick | RightClick

data MousePosition = MkMousePosition Float Float

mouseClickEvent :: Event MouseClick
mousePosition :: Behavior MousePosition

sampleWith :: Event a -> Behavior b -> Event b
filterEvent :: Event a -> (a -> Bool) -> Event a
accumulate :: a -> Event (a -> a) -> Behavior a

mapEvent :: (a -> b) -> (Event a -> Event b)
mapBehavior :: (a -> b) -> (Behavior a -> Behavior b)
-- Event and Behavior can be instances of Functor.

leftClickedPoint :: Event MousePosition
leftClickedPoint = sampleWith
    (filterEvent mouseClick isLeftClick)
    mousePosition

where
    isLeftClick :: MouseClick -> Bool
    isLeftClick LeftClick = True
    isLeftClick _ = False

leftClickHistory :: Behavior [MousePosition]
leftClickHistory = accumulate [] consMousePosition

where
    consMousePosition :: Event ([MousePosition] -> [MousePosition])
    consMousePosition = (mapEvent (:) leftClickedPoint)
    (: ) :: a -> [a] -> [a]

heatMap :: Behaviour Screen
heatMap = mapBehavior clicksToHeatmap leftClickHistory

clicksToHeatmap :: [MousePosition] -> Screen
Listing 2.4: FRP server

```
-- FRP Combinators
filterJust :: Event (Maybe a) -> Event a
filterRight :: Event (Either a b) -> Event b
filterLeft :: Event (Either a b) -> Event a

sampleWith :: Event a -> Behavior b -> Event b
accumulate :: a -> Event (a -> a) -> Behavior a

(<*> :: Applicative f => f (a -> b) -> f a -> f b
<$> :: Functor f => (a -> b) -> f a -> f b

appBA :: Behavior (a -> b) -> Event a -> Event b
appBA b e = (sampleWith e b) <*> e

lastEvent :: a -> Event a -> Behavior a
lastEvent initial event = accumulate initial (const <$> event)
const :: a -> b -> a

{- rawRequestsE construct provided natively by the language -}
type RawRequest = String
rawRequestsE :: Event RawRequest

-- App Code
data User = MkUser { id :: String }
type RegisteredUsers = Set User
data Request = MkRegister User
| MkUnregister User
parseRawRequest :: RawRequest -> Maybe Request

-- For every register request, it is either
-- a register error or
-- a new set of registered uses.
updateUsers :: RegisteredUsers -> Request ->
         (Either String RegisteredUsers)

-- FRP Combinators & App Code
parsedRequestE :: Event Request
parsedRequestE = filterJust (parseRawRequest <$> rawRequestsE)

registerResultE :: Event (Either String RegisteredUsers)
registerResultE = updateUsers <$> registredUsersB 'appBA' parsedRequestE

registredUsersB :: Behavior RegisteredUsers
registredUsersB = lastEvent Set.empty (filterRight registerResultE)

responses :: Event String
responses = filterLeft registerResultE
```
will be addressed in 3.1.

The two above aspects require:

- Ability to generate or compile suitable implementations through analysing the semantics of the code.

- Ability to extend the language with primitive/native features to fit the particular domain (e.g. for web apps a native "Event RawRequest" and a handler for "Event Response")

Current FRP implementations are not made with these requirements in mind, discussed in subsection 4.1.2

Modular Language Stacking is proposed as a remedy to the two above requirements.
3 Modular Language Stacking

Modular Language Stacking (MLS) is proposed as a paradigm in which extensible embedded domain specific languages (EDSL) are developed in some host language. Where each EDSL is extensible to accommodate changes in the domain, and where abstract syntax trees (ASTs) of higher level EDSLs are compiled down to lower level EDSLs.

3.1 Modular Languages

Half of the idea of MLS paradigm is to have languages that can be extended with additional capabilities.

Using the example of 2.4, we say (FRP + Lambda) is our base language, containing capabilities for Behaviors, Events, combinators on the them, and also ability to define algebraic data types and to pattern match and define functions on them. The base language on its own cannot specify interaction with the outside world, therefore we extend our (FRP + Lambda) language with a FRPWebServer module, giving (FRP + Lambda + FRPWebServer).

The FRPWebServer provides access to the "rawRequestsE :: Event RawRequest" event, which denotes the machines incoming HTTP requests.

Another way to one might want to extend their language is to be able to define the client side webpage itself along with the server in a multi-tier fashion [2].

Instead of FRPWebServer, we would use a module FRPWebApp. The FRPWebApp module provides constructions to denote client side Buttons and Displays, as Events and Behaviors. This language allows us to denote web client and server logic, while keeping all of it enclosed in the pure FRP system. FRPWebApp can itself be continuously extendend with additional denotative capabilities as unforeseen requirements in the domain arise.
3.2 Language Stacking

The language stacking of MLS is to supplement and support the high level abstractions.

The second half of the questions we had in subsection 2.2.2 were "how do we attach an implementation?".

An (FRP + Lambda + FRPWebServer) language will require generated code for both the browser (JavaScript) and server (x86), hence top level languages abstract syntax tree will need to be processed and compiled down to other languages.

To illustrate existing forms of language stacking:

- 3.1 illustrates compilation as an EDSL to EDSL transformation, excerpted from [3]. A high level imperative EDSL is compiled to a lower level imperative EDSL, before being compiled to C source code. The translation from low level to C code is independent from the high level representation. The "HighExp" can be extended and altered, without needing to modify the compilation to the C code, the only change needed is to the function "translateHigh :: Prog HighExp a -> Prog LowExp a".

- 3.2 demonstrates an EDSL that generates JavaScript code, excerpted from [4].

To derive an implementation for the FRP network, multiple aspects can be taken into consideration:

- The semantics of the current FRP network specification.
- The runtime statistics and temporal characteristics of Events and Behaviours of previous instantiations of the network.
- Statistics and storage formats of the existing persisted data from previous FRP network instantiations.

The different aspects may require different completely different class of analysis and optimisations.

It may be useful to generate intermediary languages of which purpose is to be able to perform simpler syntax tree traversals and optimisations.

So the language stack can consist of:

- High-level language that the user interfaces with.
- Mid-level intermediary languages that is used to write optimisations for.
- Mid-level languages that are used to directly program parts of the high-level implementation.
Listing 3.1: EDSL to EDSL translation

```haskell
prog = do
  i <- fget stdin
  printf "sum: %d\n" $ sum $ map square (0 ... i)
where
  square x = x*x

-- Generates C:

#include <stdint.h>
#include <stdio.h>

int main()
{
  uint32_t v0;
  uint32_t let1;
  uint32_t state2;
  uint32_t v3;

  fscanf(stdin, "%u", &v0);
  let1 = v0 + 1;
  state2 = 0;
  for (v3 = 0; v3 < (uint32_t) (0 < let1) * let1; v3++) {
    state2 = v3 * v3 + state2;
  }
  fprintf(stdout, "sum: %d\n", state2);
  return 0;
}

data Prog exp a where ...

data LowExp a where ...

data HighExp a where ...

lowToCode :: Prog LowExp a -> String
translateHigh :: Prog HighExp a -> Prog LowExp a

compile :: Prog HighExp a -> String
compile = lowToCode . translateHigh
```
-- Haskell / Sunroof

```haskell
code :: JS t ()
code = do
  name <- prompt "Your name?"
  alert ("Your name: " <> name)
```

// JavaScript
```javascript
var v0 = prompt("Your name?");
alert("Your name: " + v0);
```

- Low-level languages such that low level optimisations can be performed.
- Low-level languages that are used to directly program parts of the upper layers.
- Machine level platform specific code.

We can imagine that multiple different top layer languages might be compiled down to the same intermediary representation, allowing for the reuse of the optimisations targeted at the intermediary representation. This is similar to how many languages compile down to llvm's low level intermediary representation [5], to be in turn compiled to assembly code. Though rather than having one low level intermediary representation, there can be multiple mid level ones that correspond to specific optimisation stages for the high level language.

As well as using mid level languages as an optimisation target, parts of the high level language implementation can be written in the mid level language itself. For example a mid level language might be something akin to Erlang, which is suitable for programming distributed systems.

We can imagine that generic operations on ASTs may exist across different languages. Libraries of common, generic AST processing functions can be developed. This is one of the motivations to keep all the EDSLs under the same host language.

### 3.3 Balancing Compilation and Abstract Language Features

At the time of writing the compiler for some EDSL, it might not be feasible or even possible to write an advanced enough compiler to fit the requirements of the domain.

As an escape hatch from needing advanced compilation methods, the language might need to be extended with directive capabilities, which hint the underlying compiler of the implementation. For example, a directive can be something that allows the programmer to denote that the time between an event propagating a change for another event should be...
computed within some upper bound. The compiler will have hints on what data structures
to use and where the performance tradeoffs can be made. This can be also thought of as
extending the domain according to the requirements.

Another example might be that some components in the FRP network are unecessarily
computationally expensive to keep fully reactive. An extension to the FRP language might
be created that has a construction similar to a Behavior. But unlike the semantics of a
Behavior, where the value in the Behavior must be give back the most updated value of its
derivatives, the new construction allows for immediate sampling of "stale" values which have
not yet been updated to the new values triggered by the composite Events and Behaviours
that build it up.

Another approach for specifying the runtime requirements for the app could be dealt in the
layer underneath. Based on the FRP specification, the compiler generates a user interface
which graphically displays the network, the user can then specify through the user interface
the performance requirements of the application. The compiler then saves and uses the
inputs from the interface to perform the suitable optimisations.

3.4 Large Scale MLS

A large scale adoption in MLS may see improvements in various aspects of the relationship
between software and humans.

3.4.1 Organisation Structure

It has been observed that the product an organisation produces has an isomorphism to the
communication structures of the teams that created it [6].

Cleanly isolated domains may allow individual teams to cover more surface area of a single
layer of the software. Time and energy is saved from needing to be concerned about details
below their layer. This will allow the surface of each layer to have a more uniform and
cohesive shape.

3.4.2 Swapping out and Extending Implementation

High level code can have multiple possible implementations. Features can be added on top
of the existing semantics of the language. For example, the runtime of the application can
be injected with logging capabilities without explicitly specifying so in the application
layer.
3.4.3 Specialised Organisations

A decoupling of layers can allow delegation of layers to external organisations. Specialised organisations can become developers of implementations, providing different features and guarantees:

- Logging.
- Scalability.
- Quality assurance and real time performance.
- Data gathering.
- Data privacy standards and practices.
- Data simulation and generation.
- Statistics interpretation and pattern detection.
- FRP network visualisations.

End-user focused organisations can emerge that only concern itself with the high level services of the application. Their focus might be on:

- Working closer with the customers.
- Interpreting the data that’s gathered.
- Rapidly iterating on the product.

3.4.4 Open Access Application Interface

In FRP networks, we notice interactions are defined in a logic appending manner i.e to construct a new data relation, only way is to take existing ones and combine them together.

An organisation can be responsible for compilation, execution, and data management of FRP Networks sent in by end-user focused orgs. These FRP Networks can be made open access, anyone can (or only registered orgs) can view the high level structure of them, and add to them, the resulting addition can be non-destructively integrated into production as a large single network.

Synergetically, the high level nature of the FRP DSL reduces the complexity of understanding unseen-before code written by other organisations, causing less friction to collaborate.

An extremely close integration and collaboration between multiple organisations and applications can take place.
3.4.5 Data Usage Transparency & End-User Control

An implementation extension can make it possible to propagate the FRP network configuration to additional applications for end-users to see how and where their personal data is being used in the FRP network.

The user can then opt-out of certain FRP Network pathways of their choosing. Committees for data privacy can be set up that review and categorise the FRP Network pathways into privacy levels, then a person can alternatively subscribe their data to a privacy level rather than needing to understand the FRP Network description.

The implementation only permits storage and dataflow of personal data that is agreed upon by the person.

3.4.6 Responsible Data Handling

Since the data is highly structured due to the usage of algebraic data types. Automated methods can be employed to establish differential privacy [7] procedures.

Specialised data analytics organisations can be established to whom data analysis work is contracted to, and who ensure the employment of practices that do not compromise the privacy of individuals.

Data can be generated in accordance to the statistical properties of the existing data for application testing purposes.

3.4.7 Program Stack Awareness & Security through Minimalism

It can be imagined that a DSL for Operating System specification can exist, and can be a part of the MLS stack.

All layers of stack have the possibility to be informed by each other. Operating system features can be compiled according to the needs of the application, rather than using a general purpose one. Removing unnecessary capabilities (e.g. user-oriented file systems, super users) out of the system reduces the points of entry an attacker can exploit.
4 Toward Actualising Modular Language Stacking

The requirements of a scalable MLS for the host language are to be able:

- Define multiple EDSLs of varying abstraction levels.
- Support a sophisticated user interface to the embedded language.
- Support modular definition and composition of languages.
- Support robust and reusable transformations on the languages ASTs.

Haskell has been chosen as the host language to inspect the feasibility of fulfilling the MLS requirements.

There however does exist a large amount of other options when it comes to choosing a host programming language, even languages geared directly towards defining DSLs. These options have not been explored as a part of this work, though a brief overview is given in section 7.2.

Haskell has been a vehicle for programming language research for decades, and perhaps due to its pure and typed nature, the literature often ties in the practice to category theory. This allows for ideas to be transferred to other languages, as well as keeping the ideas bundled in a unifying framework where they can be identified and distinguished from one another.

Haskell can also be seen to act as a baseline: "How far can we go without introducing exotic programming language features?"

4.1 Embedded Domain Specific Languages in Haskell

The purpose of writing domain specific languages in Haskell is provide a programming interface via functions and datatypes that reduces the conceptual complexity just to that domain.

The term "embedded" refers to the fact that the DSL is written inside a host language, i.e
Haskell. The benefits of a DSL being embedded is that all the mechanisms of the host language are at the disposal of the DSL writer. For example the EDSL writer can take advantage of the fact that they don’t have to:

- Write a parser for their language.
- Implement a type checker.
- Implement a runtime for their language.
- Compile it.

Additionally the algebraic data types in Haskell allow for very direct domain modelling. EDSLs is a rich topic in the Haskell literature and has been a topic of interest for decades.

This section:

- Highlights the relevant aspects of EDSLs for MLS. subsection 4.1.1
- Identifies key pieces of literature and techniques that orient our understanding of how close the actualisation of MLS might be and what the next steps are. section 4.2

### 4.1.1 Shallow vs. Deep Embeddings

A fundamental distinction of embedded languages is if they are shallow or deep.

Shallow embeddings use hosts language features and evaluation. Essentially they are a just collection of composable function that are conceptually high level operations on the domain.

Deep embeddings are also used as a collection of functions that are conceptually high level, except these functions, also called deeply embedded combinators are not defined such that they directly evaluate to result values of the domain, these functions first evaluate to an Abstract Syntax Tree represented by an Algebraic Data Type that represents the operations on the domain. Then in turn, an interpretation can be attached to the AST.

A number of things can be done to an AST:

- Rewriting and optimising the AST.
- Translating initial language AST into another another language AST.
- Interpreting the AST in the host language runtime.
- Translating the AST into a String that represents code. (for example C code)
A shallow embedding does not allow for inspection of how a computation was constructed, as the functions the user uses are evaluated directly to a value in the domain.

Listing 4.1 illustrates the same EDSL in both a deep and shallow setting.

The more deeply embedded the EDSL is, the fewer host language features it can piggy back off of, but the more possibilities for optimisations and compilation.

2 categorisations can be made of how deep and shallow EDSLs can interplay.

- In [8], [9], it is illustrated how an interface to the deeply embedded language can piggy back off of the host language features.

  An example being lambda binding. Rather than supporting lambda binding in the deeply embedded AST, the shallow part of the EDSL uses the host languages lambda binding mechanism to bind an AST to the LHS of a function definition. Wherever the variable occurs on the RHS, the LHS AST is spliced into that position of the RHS AST. The tradeoff of this approach is that code is duplicated when a variable on the LHS is used multiple times on the RHS.

- In [10], shallow part of EDSL is not used by the end-user of the EDSL, rather the shallow EDSL is more of an EDSL whose domain is the deeply embedded language. The shallow EDSL is used solely to define transformations on the AST constructed by the deeply embedded combinators.

section 4.2 focuses mostly on the second type of interplay, as for MLS a deep embedding is prioritised as it allows for advanced compilation opportunities.

### 4.1.2 FRP Embeddings

Current FRP implementations [11], [12], [13], lean majorly towards a shallow embedding.

The operational implementation of the network is limited to the shallow embedding semantics, and unless the shallow embedding targeted the certain execution style at the outset, the FRP implementation may be limited to:

- Executing on the runtime of a single machine.
- Having the operational execution be more or less the same shape as how the network was specified, without static rewriting of the network layout.
- Fixed sources of information to take hints on optimising the network.

In general, the shallow embedding limits it to a single interpretation, whereas in subsection 3.4.3 a possibility for multiple diverse implementations is required.
Listing 4.1: Shallow vs Deep

-- Shallow embedding
type Expr1 = Integer

-- interpretation and language defined at the same time
lit :: Integer -> Expr1
add :: Expr1 -> Expr1 -> Expr1

lit n = n
add x y = x + y

eval :: Expr1 -> Integer
eval n = n

> eval (add (add (lit 3) (lit 4)) (lit 5))
12

-- Deep embedding
-- algebraic data type is the abstract syntax tree for the EDSL
data Expr2 :: * where
    Lit :: Integer -> Expr2
    Add :: Expr2 -> Expr2 -> Expr2

-- deeply embedded combinators
lit2 n = Lit n
add2 x y = Add x y

-- here we can define arbitrary interpretations
-- e.g pretty printing the AST
eval2 :: Expr2 -> Integer
eval2 (Lit n) = n
eval2 (Add x y) = eval2 x + eval2 y

> eval2 (add2 (add2 (lit2 3) (lit2 4)) (lit2 5))
12
A deep embedding of FRP is required for having the ability to write diverse compiler optimisations and implementations. FRP combinators like (<>$>) and (<*>), and "accumulate" will need to be a part of the AST, defined as an Algebraic Data Type, such that the AST can be traversed and inspected.

Optimisations ideally would have access to not only the FRP network layout, but also the semantic properties of the functions that are applied within the Behaviors and Events.

Adapting our previous example, we have 4.2. By knowing of the monoidal nature (specifically associativity) of the inner computation (appending to a list) of the Behavior "leftClickHistory" and the nature of the "clicksToHeatmap" computation (a map and then a fold on the monoid structure), the "heatMap" Behavior can memoize the previous "Screen" values and simply compute "previousScreen <> mouseToScreen <$> newMouseHistoryEntry". Whereas a "heatMap" Behavior unaware of this optimisation will recompute the fold for the whole list every time the list is updated.

In the example the monoidal properties are explicitly stated, however on a large scale, relying on explicit usage of the properties across the whole codebase may accumulate missed optimisation opportunities.

An analysis of the program logic and semantics would be a way to automate detection of these properties. Therefore we are left with most likely requiring a deep embedding of algebraic data types, lambda calculus, and hence pattern matching. At this point it will require effort to replicate host language features into the embedded language, a solution to deeply embedded pattern matching and ADTs is described in [14], and discussed in subsection 4.2.3.

We can also imagine the importance of having reusable components to process the ASTs, as a lot of the implementations might have many of the compilation stages in common. This is discussed in subsection 4.2.2.

### 4.2 EDSL Techniques

Now that we have a grasp at what the requirements are in terms of Haskell EDSLs, different methods in the literature will be discussed that address specific aspects of the requirements.

#### 4.2.1 Modular Abstract Syntax Trees: Compositional Data Types

In order to extend the language of the deep EDSL, it is required to extend the AST, and thus the ADT that represents the AST. This has been identified as the expression problem [15], one of the solutions to this problem is known as the Data Types a la Carte(DTC).
Listing 4.2: FRP optimisation opportunity

```
leftClickHistory :: Behavior [MousePosition]
leftClickHistory = accumulate [] ((<>)) <$> pure <$> leftClickedPoint
  where
    pure         :: a -> [a]
    (<>))        :: (a -> b) -> (Event a -> Event b)
    leftClickedPoint :: Event MousePosition
    (<>))        :: [a] -> [a] -> [a]
    accumulate   :: a -> Event (a -> a) -> Behavior a

heatMap :: Behaviour Screen
heatMap = clicksToHeatmap <$> leftClickHistory
  where
    (<>))        :: (a -> b) -> (Behavior a -> Behavior b)

clicksToHeatmap :: [MousePosition] -> Screen
clicksToHeatmap mouseHistory = fold (fmap mouseToScreen mouseHistory)
  overlapScreens
  where
    fold       :: [a] -> (a -> a -> a) -> a
    mouseToScreen :: MousePosition -> Screen

overlapScreens :: Screen -> Screen -> Screen
overlapScreens = (<>)
```
approach, described in [16], and made more robust in [17], and [18].

DTC is used as a key mechanism for different types approaches for writing and processing ASTs for EDSLs, therefore the key mechanism will be presented in technical detail here.

It is illustrated well in [19], and will be presented here.

We start with an AST for the language "Expr":

```
data Expr = Val Int | Add Expr Expr

eval :: Expr -> Int

eval (Val n) = n

eval (Add x y) = eval x + eval y
```

We would like to extend the language with errors, so we add additional constructors, and this triggers large changes to our "eval" function that interprets the AST.

```
data Expr = Val Int | Add Expr | Throw | Catch Expr Expr

eval :: Expr -> Maybe Int

eval (Val n) = Just n

eval (Add x y) = case eval x of
          Nothing -> Nothing
          Just n -> case eval y of
                     Nothing -> Nothing
                     Just m -> Just (n+m)

eval (Throw) = Nothing

eval (Catch x h) = case eval x of
          Nothing -> eval h
          Just n -> Just n
```

It would be more modular if we only had to add interpretations to the new components of the language separately.

The DTC approach proposes to define the AST as separate datatypes, and instead of specifying a concrete type for the subexpression, a type variable "e" is used. Functor instances for them is defined for it specifying the locations where the subexpressions lie:

```
data Arith e = Val Int | Add e e

data Except e = Throw | Catch e e
```

```
instance Functor Arith where
          fmap f (Val n) = Val n
```

23
fmap f (Add x y) = Add (f x) (f y)

instance Functor Except where
  fmap _ (Throw) = Throw
  fmap f (Catch x h) = Catch (f x) (f h)

The data type "Fix f" causes the type variable "e" to be instantiated to the type "Fix f",
where f is the type constructor of kind "* -> *". Effectively "tying the knot" on the type
variable "e" to make it non-extendable and concrete.

The data type (f :+: g) e allows for continually composing multiple types of kind * -> *
until we decide to apply "Fix" on it.

data Fix f = In (f (Fix f))

data (f :+: g) e = Inl (f e) | Inr (g e)

type Expr = Fix (Arith :+: Except)

A generic recursion scheme [20] is defined for (Fix f) i.e fold. Functions that define a step of
recursion have the type (f a -> a).

And now the single step of the recursion can be defined for each data type individually. To
make sure the a data type has an implementation, it is type checked by usage of a type class
"Eval".

fold :: Functor f => (f a -> a) -> Fix f -> a
fold f (In t) = f (fmap (fold f) t)

class (Functor f, Monad m) => Eval f m where
  evAlg :: f (m Int) -> m Int

instance Monad m => Eval Arith m where
  evAlg (Val n) = return n
  evAlg (Add x y) = do n <- x
                      m <- y
                      return (n + m)

instance MonadPlus m => Eval Except m where
  evAlg (Throw) = mzero
  evAlg (Catch x h) = x `mplus` h

instance (Eval f m, Eval g m) => Eval (f :+: g) m where
  evAlg (Inl x) = evAlg x
  evAlg (Inr y) = evAlg y

24
eval :: Eval f m => Fix f -> m Int
 eval = fold evAlg

This method achieves modular AST definition and modular evaluator definitions, as well as
being able to define different evaluation functions for the same AST.

DTC describes smart constructors which allow to construct values of the AST without the
overhead of needing to use the Inl and Inr constructors, details won’t be provided here,
assuming smart constructors defined, expressions can be defined and evaluated as so:

three :: Fix Arith
three = val 1 ‘add‘ val 2
error :: Fix (Arith :+: Except)
error = val 42 ‘add‘ throw

> eval three :: Identity Int
   I 3
> eval three :: Maybe Int
   Just 3
> eval error :: Maybe Int
   Nothing

We can call the same evaluation function on different subsets of the language.

Applying DTC to a modular FRP language, we get:

data FRP x where
   EmptyE :: FRP x
   MergeE :: x -> x -> FRP x

   ConstB :: a -> FRP x
   AppFRP :: x -> x -> FRP x
   MapFRP :: x -> x -> FRP x

data UiFRP x where
   DisplayB :: x -> UiFRP x
   TextFieldB :: UiFRP x
   ButtonE :: UiFRP x

example :: Expr (FRP :+: UiFRP)
example = In $ Inr $ DisplayB $ In $ Inl $ ConstB "Hello"
If we observe the type signature of the example, the approach does not use the type checking capabilities of Haskell. The user of the language does not know what are the types of the expressions that they are constructing. Additionally, the possibility of ill-formed expressions, requires the EDSL writer write their own type checker.

[21] uses DTC as part of its generic AST framework. Notably this framework allows type signatures of the form "expr :: AST dom sig" where "dom" is the domain e.g FRP :+: UiFRP and "sig" is the type of the expression that the AST represents in that domain. Remediating the lack of types for the end-user.

Compositions of data types corresponds quite directly to our modular language requirements of MLS. But DTC on it’s own is not suitable as an interface for programming, so remediations are necessary so that Haskells type system can be more in use.

4.2.2 Modular Tree Processing: Modular Tree Automata

With multiple ways to combine a set of languages, slightly different variations in the processing may be required, and a lot of AST processing logic might still be the same. It might also be needed to swap out one component of a tree traversal with a different version, without altering the overall traversal. Therefore techniques for modular tree processing would help the stacking aspect of MLS of transforming EDSL to other EDSLs and rewrites.

[22], [23], [24], provides methods for processing ASTs in a modular way.

While there are many variations of methods among the listed literature, only one will be presented [22], as it will be enough to illustrate the modular capabilities.

The methodology directly ties in with the DTC approach.

Starting from a similar setup:

newtype Term f = In (f (Term f))

data Sig e = Val Int | Plus e e

instance Functor Sig where
    fmap f (Val i) = Val i
    fmap f (Plus x y) = Plus (f x) (f y)

cata :: Functor f => (f a -> a) -> Term f -> a
cata f (In t) = f (fmap (cata f) t)

evalAlg :: Sig Int -> Int
evalAlg (Val i) = i
evalAlg (Plus x y) = x + y

eval :: Term Sig -> Int
eval = cata evalAlg

The paper highlights that the "cata" (called "fold" in the previous section) corresponds to a
Deterministic Bottom-Up Tree Acceptor in the Tree Automata literature:

type UpState f q = f q -> q
runUpState :: Functor f => UpState f q -> Term f -> q
runUpState = cata

The paper gives an example of compiling our Sig e language into code for a virtual
machine:

type Addr = Int
data Instr = Acc Int | Load Addr | Store Addr | Add Addr
type Code = [Instr]

We define the f-algebra that implements a translation from Sig to Code. An f-algebra, in this
context is name for the thing of the type "f a -> a", and in this case we have "Sig (Code,
Addr) -> (Code, Addr)", and with the type synonym, "UpState Sig (Code, Addr)"

The important observation is that we are threading a piece of intermediary state throughout
the computation, i.e the Addr. For this computation we need to know the height of the
expression, computed by "h = 1 + max lh rh".

The final value that we care about is "Code"

codeAddrSt :: UpState Sig (Code, Addr)
codeAddrSt (Val i) = ([Acc i], 0)
codeAddrSt (Plus (x , lh) (y, rh))
  = (x ++ [Store rh] ++ y ++ [Add rh], h)
  where h = 1 + max lh rh

code :: Term Sig -> Code
code = fst . runUpState codeAddrSt

Ideally we would be able to define the height computation component individually:

heightSt :: UpState Sig Int
heightSt (Val _) = 0
heightSt (Plus x y) = 1 + max x y

A notion of a state space is encoded on the type level, which corresponds to the available
computations that can be extracted during a step of an f-algebra. "a :e: b" can be read as
"a is an element of state space b", and if this type level property holds, then we define how to extract the value out of the state space via the "pr" function:

```haskell
class a :e: b where
    pr :: b -> a
instance { -# INCOHERENT #-} a :e: a where pr = id
instance { -# OVERLAPS #-} a :e: (a, b) where pr = fst
instance { -# OVERLAPS #-} (c :e: b) => c :e: (a, b) where pr = pr . snd

type DUpState f p q = (q :e: p) => f p -> q
dUpState :: Functor f => UpState f q -> DUpState f p q
dUpState st = st . fmap pr

runDUpState :: Functor f => DUpState f q q -> Term f -> q
runDUpState = runUpState

We update our tree acceptor to be able to hold a state with multiple types (in otherwords receive a polymorphic tuple of any nested length), given by the type signature "(q :e: p) => f p -> q"

Now we declare in the type signature of the f-algebra that this computation requires that a value type of Int (i.e the height of expression) is required to be in the state space.

The value can then be extracted with a "pr", the type inference specifies which value to pick out:

```haskell
codeSt :: (Int :e: q) => DUpState Sig q Code
codeSt (Val i) = [Acc i]
codeSt (Plus x y) = pr x ++ [Store a] ++ pr y ++ [Add a]
    where a = pr y
```

The limitation here is that the state space can only contain unique types, but this limitation is alleviated in the other variants tree automata the paper describes.

For completeness combinator for combining state space is:

```haskell
(x) :: (p :e: c, q :e: c) => DUpState f c p -> DUpState f c q -> DUpState f c
sp x sq = \t -> (sp t, sq t)

code' :: Term Sig -> Code
code' = fst . runDUpState (codeSt x dUpState heightSt)
```
The more relevant part is that if we want to extend our AST, this method also supports the DLC approach.

For example, we want to extend our language with an increment operation, we then provide an implementation for "heightSt" for that individual datatype:

```haskell
data Sig e = Val Int | Plus e e
newtype Inc e = Inc e
type Sig' = Inc :+: Sig

class HeightSt' f where
    heightSt' :: UpState f Int

instance (HeightSt' f, HeightSt' g) => HeightSt' (f :+: g) where
    heightSt' (Inl x) = heightSt' x
    heightSt' (Inr x) = heightSt' x

instance HeightSt' Sig where
    heightSt' (Val _) = 0
    heightSt' (Plus x y) = 1 :+: max x y

instance HeightSt' Inc where
    heightSt' (Inc x) = 1 :+: x

height :: (Functor f, HeightSt' f) => Term f -> Int
height = runUpState heightSt'
```

This approach not only allows us keep different functionalities of f-algebras separate and allow for it’s individual reuse in multiple other f-algebras, but also allows us to extend the datatypes that they operate on without needing to change the logic of the original datatype f-algebras.

This technique seems promising, and looks like it can be a major part of the scaffolding for MLS.

### 4.2.3 User Defined Data Types and Pattern Matching: Embedded Pattern Matching

As discussed in subsection 4.1.2, it will be necessary support a deep embedding of a lambda calculus with algebraic data types, and along with that, it would be desireable to be able to use pattern matching as the user interface. However pattern matching is quite a fundamental part of the host language, and making it a part of the deep embedding requires that pattern matching somehow be recorded into the AST of the expressions. [14] shows a
method to translate pattern matching and algebraic data types in the host language to their deeply embedded versions in the target language. Resulting in a native-feeling user interface for defining functions in the deep EDSL:

```hs
  toSKI :: Exp Lambda -> Exp SKI
  toSKI = match \case
    Var_ v -> . . .
    App_ f x -> . . .
    Lam_ v x -> . . .
```

The embedded language contains primitives for tuples, and it’s with these nested tuples the embedded language is able to encode algebraic data types. The paper uses generics [25] to translate ADTs from the host language into the embedded language as a nesting of tuples. The ADT needs to be an instance of a typeclass "Elt", though the typeclass is generically derivable.

```hs
data Exp a where
  Const :: EltR a -> Exp a
  Tuple :: Tuple (TupleR t) -> Exp t
  Prj :: TupleIdx (TupleR t) e -> Exp t -> Exp a
  ...

data Tuple t where
  Unit :: Tuple ()
  Exp :: Exp a -> Tuple a
  Pair :: Tuple a -> Tuple b -> Tuple (a, b)
  ...

data TupleIdx s t where
  PrjZ :: TupleIdx t t
  PrjL :: TupleIdx l t -> TupleIdx (l, r) t
  PrjR :: TupleIdx r t -> TupleIdx (l, r) t

class Elt a where
  type EltR a
  fromElt :: a -> EltR a
  toElt :: EltR a -> a
  ...
```

Example of manually lifting native ADT to deep encoding:

```hs
data Point = Point Float Float
data V2 a = V2 a a
```
Every time a data type is eliminated from a product type, it is in reality adding onto the deeply embedded AST. The change of types specified in the GADTs on the projections of tuples immitate eliminations on the tuples on the type level, but in reality the projection gets recorded on the AST.

```haskell
liftPoint :: Point -> Exp Point
liftPoint (Point x y) = Tuple $ Unit (Pair
    (Pair Unit (Exp Const x))
    (Exp (Const y)))
-- or = Const . fromElt

-- example of a projection thats actually adding onto the AST
xcoord :: Exp Point -> Exp Float
xcoord p = Prj (PrjL (PrjR PrjZ)) p
```

The pattern synonyms GHC extension provides a way to disguise a pattern match into some other operation, and it’s with this mechanism the edsl achieves the illusion of native pattern matching. As well as defining how destruct a value "matchPoint", pattern synonyms allow for defining synonyms for constructors as well "buildPoint".

In "matchPoint" when pattern matching on an embedded data type p, we are defining the corresponding left and right projection to be a tuple where the left value is an AST that denotes a left projection, and the right value is an AST that denotes a right projection:

```haskell
pattern Point_ :: Exp Float -> Exp Float -> Exp Float
buildPoint :: Exp Float -> Exp Float -> Exp Point
buildPoint x y = Tuple $ Unit (Pair (Pair Unit (Exp x)) (Exp y))

matchPoint :: Exp Point -> (Exp Float, Exp Float)
matchPoint p = (Prj (PrjL (PrjR PrjZ)) p, Prj (PrjR PrjZ) p)

pattern Point_ x y <- (matchPoint -> (x,y))
  where Point_ = buildPoint
```

Embedded pattern synonyms for all product types work in the same way can abstract into a polymorphic pattern synonym:

```haskell
pattern Pattern :: IsPattern s r => r -> Exp a
pattern Pattern r <- (matcher -> r)
  where Pattern = builder
```
class IsPattern s r where
    builder :: r -> Exp s
    matcher :: Exp s -> r

These pattern synonym definitions are automated through template haskell. Sum types require special treatment, which will not be summarised here.

This work reveals the that deep EDSL can be made to immitate the language features of Haskell, without sacrificing the user interface. A hinderance might be that the encodings of the embedded ADTs is not as straightforward as one might like, which might make it difficult to define AST transformations.
5 Evaluation and Conclusions

In the first sections of this work, using Functional Reactive Programming, an approximation and envisionment was developed about what an ideal, deliberate, and conscious software situation would look like. The vision was strong enough to grasp what the requirements would be, and were expressed through the medium of Haskell EDSLs.

The requirements lead to hypothesising that it would consist of possibly high level, extensible, deeply embedded domain specific languages, with modular ways to define AST transformations and optimisations on them, so that the components can be reused as libraries.

A select few of the approaches from the Haskell literature were demonstrated and reviewed. On their own, the approaches seem to solve for some of the fundamental requirements of MLS, and serve as a good starting point for further exploration.

A part of the review was to investigate how simple the host language can be to support it, and we saw that Haskell was being stretched to its limits with various GHC extensions and type level gymnastics. Furthermore it is not entirely clear how the individual pieces can fit together under one scaffolding.
6 Future Work

The scope of this work did not include implementing a deeply embedded FRP DSL, though using the DSL techniques discussed, it could be fruitful to attempt it as it does not seem like a deep embedding exists. Also, it would allow us to observe how the pieces of the techniques can fit cohesively together, if at all.

Furthermore using the deeply embedded FRP DSL, the things that can be explored are:

- Various transformations that can be made on the FRP graphs.
- Underlying representations and languages that can express the FRP graphs.
- Compilations of the FRP graphs to efficient low level implementations.
- Domain extensions to the FRP that, for example, allow to define a totally pure web server.
7 Related Work

7.1 Haskell and EDSLs

- Some examples of EDSLs in Haskell are Feldspar [26] for performant imperative programs, and Accelerate [27] for GPU programming. Feldspar uses the Syntactic library [28] for general purpose AST definition and utilities. The newer versions of Accelerate include an implemented embedding pattern matching [14].

- Monad Reification [29] describes how the monad combinators can be defined as ASTs such that do notation is available, and furthermore how it can be compiled into instructions for a domain, while preserving the monad laws.

- Trees that Grow [30] and Final Tagless [31], are other approaches for tackling the expression problem, which have not been explored in this work.

- Quasiquoting, [32], [33], are approaches in Haskell to run compile time computations on the Haskell AST to generate Haskell code, in a reminiscent fashion to lisp macros. The utility towards MLS has not been explored in this work.

7.2 Host Languages

- The Racket Manifesto [34] describes an extremely similar ethos to Modular Language Stacking, in which it reiterates how Racket is a language to write languages, having the consequence that the Racket programs are interconnected and multilingual.

- Dependently typed languages such as Idris[35], Agda[36], can be though of having a type system which is more far reaching than Haskells, and we have seen the need to stretch Haskell type system in the different techniques. As well as that the lanaguages have features intended to support the writing of domain specific languages, including reflection [37], [38].

- reFlect [39] is a strongly typed ML-like language with quoatation and anto-quatation mechanism, which may be useful for deep EDSLs, though the langauge is targeted
hardware specification and verification.

- MetaML [40] is an example of one of the multistage programming languages, made for compiling code in multiple stages and generating type-safe programs.
Bibliography


