

Towards Modular Language Semantics of WebDSL: A Case Study of Using Algebraic Effects in Haskell for Language Specification

Master's Thesis



Aleksandra Wolska

Towards Modular Language Semantics of WebDSL: A Case Study of Using Algebraic Effects in Haskell for Language Specification

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Aleksandra Wolska
born in Warsaw, Poland



Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

© 2024 Aleksandra Wolska.

Cover picture: Haskell logo and three versions of subtyping infix used in the framework.

Towards Modular Language Semantics of WebDSL: A Case Study of Using Algebraic Effects in Haskell for Language Specification

Author: Aleksandra Wolska
Student id: 4950836
Email: A.K.Wolska@student.tudelft.nl

Abstract

WebDSL is a DSL for creating web applications, combining many different aspects and domains of web design in a single language. The dynamic semantics of this language are not defined, despite multiple attempts, abandoned due to complexity of the language and lack of expression of chosen frameworks. We adapt the algebraic effects and handlers approach and the framework introduced in *Datatypes a la carte* (Swierstra, 2008) to create a modular denotational semantics model of WebDSL, extending the framework by a bifunctor formulation for multi-sort syntax definition that allows us to distinguish between effects raised by different components of the language. In the process of defining the framework and semantics in Haskell, we encountered obstacles and of working with algebraic effects and handlers paradigm in the language, leading us to compile workarounds, solutions and pitfalls to avoid when constructing and maintaining such model. In evaluation of the framework approach with earlier attempts at defining dynamic semantics for WebDSL, we find algebraic effects and handlers to be a viable and successful approach for modelling a rich DSL such as WebDSL, and propose possible improvements to the WebDSL compiler.

Thesis Committee:

Chair: Dr. A. Katsifodimos, Faculty EEMCS, TU Delft
Committee Member: Dr. J. Cockx, Faculty EEMCS, TU Delft
Committee Member: Dr. C. Poulsen, Faculty EEMCS, TU Delft
University Supervisor: Dr. D. Groenewegen, Faculty EEMCS, TU Delft

Contents

Contents	iii
List of Figures	v
List of Tables	v
1 Introduction	1
1.1 Goals of the Project	4
1.2 Contributions	5
2 Framework by Example: Modelling WebDSL Functions	7
2.1 Pipeline of the Model	7
2.2 WebDSL Programs in Haskell	8
2.3 Modular Semantics	13
2.4 Denotational Mapping	15
2.5 Handling Effects	20
3 WebDSL components	27
3.1 Definitions	27
3.2 Global variables	30
3.3 Templates Component	32
3.4 Multi-phase Processing	39
4 Evaluation of the Algebraic Effects and Handlers Framework Approach in Haskell	41
4.1 Reusable Effects	41
4.2 Composite Handlers	45
4.3 Applying Higher-order Effects in the Framework	46
4.4 Stronger Type System of a Dependently Typed Language	48
4.5 Tooling	49
5 Case study of WebDSL: Evaluation	51
5.1 Comparison with Other Approaches to Model WebDSL's Dynamic Semantics	51
5.2 (Lack of) Modularity in the Denotation	55
5.3 Possible Improvements to WebDSL	57
5.4 Evaluation of the model	57
6 Related work	61
7 Conclusion	63

7.1 Future Work	63
Bibliography	65
Acronyms	69
A Effects used in the model	71
A.1 Functions' component's effects	71
A.2 Templates' component's effects	72
B WebDSL syntax modules	75
B.1 Functions' component syntax	75
B.2 Templates component	78
B.3 Definitons	80

List of Figures

1.1	Rendered HTTP Response from WebDSL app	2
1.2	Rendered HTTP Response to non-validated request	2
2.1	Pipeline of the model	7
2.2	The symmetric(left) and asymmetric (right) abstract syntax trees (ASTs)	11
3.1	Pipeline including definitions	28
3.2	Pipeline including Global Variables	31
3.3	Request processing modes	40

List of Tables

5.1	Approximate lines of code (LOC) of the denotational models	52
5.2	Semantics coverage between the models	53
5.3	WebDSL language concepts coverage	58

Chapter 1

Introduction

WebDSL is a domain-specific language (DSL) for creating web applications, compiled to Java applications (D. M. Groenewegen, Chastelet, Krieger, and Pelsmaeker, 2023; D. Groenewegen, 2023). Where web design is known for requiring multiple DSLs to cover all standard and additional features, WebDSL is built to integrate them and provide abstractions to increase reusability and security in the final product.

WebDSL provides an innovative combination of both standard web server operations and usually unintegrated solutions:

- custom HTML templating semantics encapsulating a small general purpose language
- an abstraction of database access
- an abstraction of database access and sessions management
- request processing lifecycle
- a novel abstraction of secure user input handling
- integrated AJAX actions
- integrated access control semantics

Compared to other full stack frameworks, it is unique in that it does not build on top of a general purpose language that would constraint the design choices. Instead, with a custom compiler, it allows for providing stronger integration guarantees and thus more security by construction (D. Groenewegen, 2023).

WebDSL programs typically consist of a list of various definitions, with the `page root` typically being the entry point of the program; the compiled web server calls these definitions where applicable. To give a taste of the capabilities of WebDSL, we present below an example subpage of a web application; it showcases an interactive user form.

```
1  entity Project {
2    name : String (id)
3    apps : {Project}
4    validate(name.length() > 0, "Project name may not be empty")
5  }
6
7  var wl      := Project { name := "weblab" }
8  var re      := Project { name := "researchr" }
9  var webdsl := Project { name := "WebDSL", apps := {wl, re} }
10
11  page editProject (p:Project) {
12    form {
13      header { output(p.name) }
```

```
14   for (app in p.apps) { block { input(app.name) } }  
15   submit action {  
16     for (app in p.apps) { app.save() }  
17   } { "save" } }  
18 }
```

First, the entity `Project` defines an interface with the database. This entity has two fields, a name and `apps`: a list of sub-projects. It also holds a validation rule for the name field. Second, three global variables defined on lines 7-9 provide instances of the `Project` entity.

The `editProject` page defines a page with an input form that allows to rename all sub-projects of a single project. The form has an input field for each of the sub-projects, and a button (`submit`) with *action* of saving the changes of the edited projects to the database. When called with `weblab` as input parameter, the server of this application will generate HTML code that a web browser renders to the output in Fig. 1.1.

WebDSL

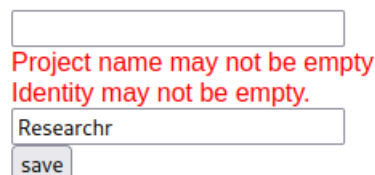


```
weblab  
researchchr  
save
```

Figure 1.1: Rendered HTTP Response from WebDSL app

It is important to notice that there is no separation between the layout, model and control elements. The application generates both server actions and client response from the same definition, managing the linking of these elements for the user. Additionally, when a client submits some data via this form, the server validates all the edited entities' properties with the pre-defined validation rules, and executes the actions based on user input. If any of the validation rules did not pass, the defined action is not executed: submitting a form with empty name will result in an error, as can be seen in Fig. 1.2.

WebDSL



```
Project name may not be empty  
Identity may not be empty.  
Researchchr  
save
```

Figure 1.2: Rendered HTTP Response to non-validated request

WebDSL has multiple projects written in it running, for example `Weblab`¹ and `Conf Researchchr` (D. M. Groenewegen, Chastelet, Krieger, Pelsmaecker, and Anslow, 2023). Initiated as a research project, it is continuously under development, with possible new directions, improvements and extensions considered. The compiler codebase is extensive and contains

¹<https://eip.pages.ewi.tudelft.nl/eip-website/weblab.html>

a large directory of unit tests. However, when improving the implementation of existing features, these unit tests might not be sufficient. This is an issue especially when altering the legacy code of which some critical features, as it might impact the security of existing applications. In order to improve the quality assurance when developing these features, more thorough documentation and verification methods should be developed.

The language does not have its *dynamic semantics* specified: the formal model of *how* the programs should behave. Such model could fill in the role of additional documentation and verification. Multiple approaches to specify the dynamic semantics have been taken so far (D. Groenewegen, 2023, p. 215), with tools, or metalanguages, such as Redex (Felleisen, Findler, and Flatt, 2009) and with Haskell, using the state monad. These attempts to model the dynamic semantics however never did come to a completion, due to the effort required to maintain them along the language or insufficient expressiveness of the modelling languages.

Defining the semantics would have multitude of benefits. Groenewegen (2023) specifies: “Besides a clean description of the behavior of WebDSL programs, such a semantic description enables automated testing of the compiler. Semantically correct test programs can be generated, and the output of the model implementation can be checked against the actual compiler implementation” (p. 215).

Based on this statement, the benefits can be organized as follows: firstly, it would provide a more understandable and concise documentation, both for users of the language and for developers trying to understand or reproduce the workings of WebDSL. Secondly, it would allow generating aforementioned regression tests with property-based testing tools. Thirdly, it would give a basis to simplify the effort needed to alter and improve the implementation of critical components, by giving a clear encapsulation of the implementation of each component, and its interaction with other components. Finally, in the principle of language adjustment based on evolution (D. Groenewegen, 2023, pp. 124-132), it could guide further language design, exposing semantics that could be abstracted more concisely.

We shall highlight here that the task we are discussing has already been tried and failed before; in the first place, we should summarize these efforts and find *where* they failed. From the two attempts, the first one applied Haskell, a general purpose functional programming language, embedding the semantics in a state monad for describing the different side effects. This approach was found to not be flexible enough, as the subset of semantics covered grew. The second attempt was centered around Redex, a DSL designed with purpose of modelling dynamic semantics (Felleisen, Findler, and Flatt, 2009). This language was found to not be expressive enough.

Groenewegen says (p. 215): “One problem we experienced is that the more details are included in this model of WebDSL, the more work it is to maintain” (p. 215). Indeed, when giving a model, we should not forget that it should be a *reduction* and that it should be *pragmatic*: namely, the model should be simpler than the system under modeling, and it should be constructed for the purpose there were made to perform (Podnieks, 2018; Stachowiak, 1974). A model lacking these features can become indistinguishable from the system under modeling, performing the same tasks, but worse, slower, and less efficiently.

With these principles in mind, we can start exploring the possible solutions. The important aspect of our task is that we are trying to *capture* existing semantics of the language, instead of simply designing them from scratch, as the suggested use cases for Redex (Felleisen, Findler, and Flatt, 2009).

Here is the place to emphasize WebDSL’s modular nature, where each component has its own set of computational effects, which might still get expanded or reformed in the future. Such cases were found to be a common pitfall when trying to give a monolithic semantics definition. The general idea for the solution is to introduce *auxiliary notation*, making the definitions composable and reusable (Benton, Hughes, and Moggi, 2002; Mosses, 1990). Ex-

plored examples of such notation include, among others, semantic models such as monads and monad transformers.

Returning to the question of pragmatism: the expected use cases were already defined, but what realization would suit them the most? The chosen framework must provide clarity to be of use as documentation, divided into components that capture the modules of WebDSL, and easily extensible, such that it can be updated alongside the language.

Algebraic effects and handlers (Plotkin and Power, 2002; Plotkin and Pretnar, 2009), the increasingly popular alternative to monads, is a framework providing this modularity and clarity. Its biggest benefit is the division of the program into the semantics interface and implementation, giving a composable overview of the language with a runnable backend. With easily replaceable handlers, it is possible to define different behavior under different environments, without altering such defined interface. The way we plan to utilize effects is not the standard approach; instead of using algebraic effects to represent *side effects* of the program itself (such as IO operations), we use them to represent the side effects of the language under modelling.

The reduction required by Stachowiak’s model theory (a model should be a reduction of the system under modelling) (Podnieks, 2018; Stachowiak, 1974) is also clear from the expected usage: what concerns us is the behavior of *correct* programs, thus focusing on completeness over soundness, and leaving programs failing static analysis as undefined. The effects interface should be built with a limited syntax to clearly expose the semantics.

There are many tools to choose from for the task of defining denotational semantics. They can be given even as plain text notation, even though that limits their applications. Some existing DSLs exist specifically for that purpose, but do not lend the power of algebraic effects, like Redex. Finally, among languages that can work with algebraic effects, there is still plenty with no clear winner.

As we are the most familiar with them, we considered both Agda and Haskell. While Agda’s power of dependent types might lend some elegant solutions, it also requires more infrastructure due to its *strict positivity* requirement. This additional boilerplate would both make it more difficult to encode the model by us and later be easily understandable to others. In the end, we decided to use Haskell as it has a much wider user base than any of the previously mentioned tools, it is a much more well-known language and thus easier to understand for potential future users of the model. We further discuss the potential of using Agda for such project in Section 4.4.

There are also several languages with native support for algebraic effects, such as Koka (Leijen, 2023), Effekt (J. I. Brachthäuser, Schuster, and Ostermann, 2020), or Frank (Lindley, McBride, and McLaughlin, 2017). We decide against using any of these languages for the framework, for two main reasons. First, they do not provide the flexibility to alter the algebraic effects and handlers implementation when necessary, as it is possible with a definition in Haskell. Such flexibility is useful when we study the effects themselves. Second, they do not have the same rich amount of libraries that were built for Haskell, which both might make writing the handlers for the framework more difficult, and limit the future uses of the framework. Haskell’s property-based testing library QuickCheck (Claessen, 2024) is especially important; none of mentioned languages provides support for this functionality.

1.1 Goals of the Project

The main research question is “Can we use algebraic effects and handlers framework to develop formal semantics for a composition of different subsets of WebDSL?” This includes the following subquestions:

- How much of WebDSL can we cover within the model?
- To what extent can we employ the effects paradigm to make the model modular, such that the components are independent of each other?
- How does the algebraic effects and handlers framework compare to the previous attempts of modelling WebDSL's dynamic semantics, in terms of maintainability and conciseness?
- Does these semantics give rise to any possible improvements of WebDSL?

We give an answer to this questions in Section 5.4

1.2 Contributions

The main contribution of this thesis is the denotational semantics model of WebDSL, which includes the following components:

- An algebraic effects and handlers framework for the semantics, adjusted to the specifics of WebDSL (Chapter 2), including its multi-layered language and multiple modes of processing (Chapter 3)
- The denotation itself, covering WebDSL's functions, entities and templating semantics, extended with processing definitions and global variables
- An evaluation of chosen implementation of algebraic effects and handlers approach (Chapter 4)
- An evaluation of algebraic effects and handlers framework for defining denotational semantics of WebDSL (Chapter 5)

The document is structured as follows: first, we give an overview of the basic algebraic effects and handlers framework, used throughout the program, in Chapter 2. Then, we describe the extensions to the framework necessary to model different components of WebDSL in Chapter 3. In Chapter 4 we discuss the drawbacks and alternatives to the used algebraic effects Haskell implementation, and in Chapter 5 we provide an evaluation of the application of algebraic effects and handlers for giving WebDSL semantics. In Chapter 6 we present related work and conclude in Chapter 7.

We assume background knowledge of functional programming, Haskell, and web programming on behalf of the reader. The background of algebraic effects and handlers is explained in Chapter 2.

Artifact

The artifact accompanying this thesis contains the dynamic semantics model of WebDSL. The syntax and denotation of different WebDSL subcomponents as divided in Appendix B, effects as specified in Appendix A, handlers, and multiple phrasings of boilerplate for different subsets of WebDSL can be found in `src` directory. The `test` directory contains unit tests making use of the different boilerplate formulations.

The artifact is available online at <https://github.com/odderwiser/webdsl-model>.

Chapter 2

Framework by Example: Modelling WebDSL Functions

In order to provide easily readable and maintainable semantics for such a complex language as WebDSL, modularity is the key necessity of our system. To freely compose different components of WebDSL and their semantics, we require a modular *syntax* construct, modular *semantics* construct and a modular *mapping* between them.

We need this setup in order to separate different components of WebDSL such that the semantics of these components can be encapsulated and presented independent of each other. This way, the complexity of the remaining components of WebDSL will not get in the way of understanding the current component if they are not relevant. In case two components *do* interact, this is also should be emphasized more clearly. The modular nature also provides us with reusability in case the semantics of only one of WebDSL components change.

In this chapter, we present the basic principles we use to build a framework for the functions component of WebDSL. In Section 2.1 we introduce the pipeline of the model, and in each of the following section, we present each consecutive step of the pipeline. While the principles of the framework are general and reappliable, for every new *layer* or WebDSL semantics, the framework requires slight adaptations and extensions in order to fit the purpose accurately. We present these adjustments in the following Chapter 3.

2.1 Pipeline of the Model

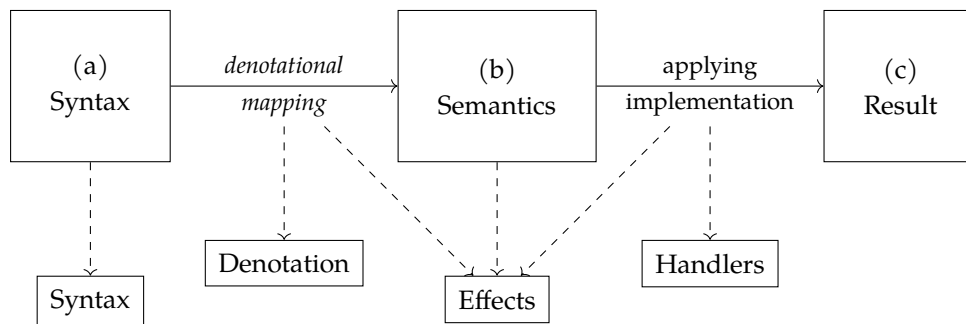


Figure 2.1: Pipeline of the model

In Fig. 2.1 we present the model pipeline that provides us with that modularity. The pipeline consists of three steps (a-c) and two transformations between them. To provide building blocks for this pipeline, we use four types of components: Syntax, Denotation, Effects and Handlers, each of them relevant only for some part of the pipeline, as denoted

by the dashed arrows. Each of these components is a Haskell module containing relevant datatypes and functions.

First, we have some WebDSL program (a). This program is not an exact syntax of WebDSL, but any correct WebDSL program can be parsed into this AST representation embedded in Haskell. We assume the *correctness* condition to be the program passing static verification. Further, in some cases, we require annotations on the AST that the static verification generates, such as type annotations or list of names in scope. Using the different Syntax components, we can combine them into various subsets of WebDSL based on what is needed in the program we want to write. The details of how we define this composition for modular syntax can be found in section 2.2.

Second, we need modular semantics (b) embedded in an intermediate representation (IR) and internally represented by a different AST, which is implemented using a Free monad. However, what is a Free monad? How do we use it for modularity? We use the Effects components to abstractly represent the various operations and their possible computational effects available in WebDSL. How exactly do we do that, and what really are algebraic effects? These and other questions will be answered in section 2.3.

Finally, we need a modular mapping function between these two ASTs. We use building blocks from the Denotation component for that purpose; the details of how we realised this can be found in section 2.4.

Since we already receive dynamic semantics in (b), we could simply stop at that point of the pipeline. However, if we can to *run* these semantics and verify the result we achieve, we are also able to apply some automatic verification. To a lesser extent, it allows us to verify the correctness of our semantics in a much easier manner. To a bigger extent, if we assume our semantics to be correct with regard to the definition of the language, we can use them to verify the correctness of the WebDSL compiler. This requires another mapping between (b) and (c), in which we apply *Handlers* that transform the Effects in the semantics into concrete Haskell operations; we can say that where Effects are interfaces of operations, Handlers are implementations of these interfaces. We will discuss our framework for this mapping in section 2.5.

The result in (c) consists of the pure value that is the outcome of the denotation function, as well as various data that is an outcome of running one of the handlers, appended to the end value by the handler.

Most of the setup discussed in this section is borrowed and altered from Swierstra (2008) and Bach Poulsen (2023). We assume familiarity with Haskell.

2.2 WebDSL Programs in Haskell

In this section, we will focus on the first element of the pipeline presented in Figure 2.1. This section will be loosely reinventing and summarizing the findings of Swierstra (2008).

First, let's consider what do we mean by Syntax components. Every module is built around some Syntax unit that could be conceptually isolated from the domain of WebDSL functions. To present the syntax used, we will present two small modules from the expressions chapter: `Arith` and `Boolean`. These modules give operations expected to be encountered in any general-purpose language, allowing us to present our framework without requiring background knowledge about WebDSL.

2.2.1 First attempt: Syntax Components as Datatypes

Here we will present a first attempt to modular syntax, how to compose it and why it does not work.

```
1 data ArEnum = Add | Div | Sub | Mul
```

```

2
3 data Arith = LitAr Int | OpAr ArEnum Arith Arith

```

Arith datatype is meant to give the basic arithmetic constructs. It has two constructors: `LitAr` for literals, which in this case means integers only, and `OpAr`, which is a general constructor for binary operations, parametrized by the `ArEnum` enum. We decided on this approach of parametrized constructor for binary operations to improve the reusability - adding a new arithmetic operation requires only adding a new constructor to `ArEnum`, the `Arith` datatype can remain unchanged.

```

1 data BEnum = Or | And
2
3 data Boolean = LitB Bool | OpB BEnum Boolean Boolean
4 | If Boolean Boolean Boolean

```

The second datatype, `Boolean`, gives the usually expected constructs for logic operations. It is build alike the `Arith` datatype, except for the additional `If` constructor for the ternary *if/then/else* operation. We do not expect that the language might need more logical ternary operations, so we do not apply the parametrized approach from binary operations.

Value Composition

Now comes the question: how do we correctly bundle together these types?

The sum type of ordinary values in functional programming is known as the **Either** type. It encapsulates many possible values, while holding only one of them at runtime. We could use it to construct a type `Either Boolean Arith` to create an expression that could be *either* of these datatypes, but holding only one of them at runtime.

However, one can imagine that with increasing the number of Syntax modules allowed in our expression, indefinitely nesting the **Either** type will be tedious to read and write; we would like to use some syntactic sugar to express that instead.

```

1 infixr \/
2 type u \/ v = Either u v

```

The infix `\/` is right-associative, which means we can write a type such as `Arith \/ Boolean \/ Expr \/ ...` and have it internally represented in a deterministic way: the leftmost datatype is always the least nested.

Why Does this Approach Not Work?

We must stop here for a while and think about this setup. As long as we only consider binary operations, these definitions might be making sense: arithmetic operations should only accept arithmetic arguments and so on.

However, introducing the ternary conditional brings some issues. We cannot express `if True then 2 else 3` with our limiting datatypes. We also cannot use the conditional as an argument for addition. If we later add a syntax module giving variables and operations on them, another sensible and expected language feature, we can see that this way of defining binary operations holds no merit.

2.2.2 Second Attempt: Syntax Components as Functors

In the second attempt, we would like the Syntax components to be parameterised such that the recursive elements of the datatype are not fixed.

```

1 data Arith e = LitAr Int | OpAr ArEnum e e
2 deriving Functor

```

```

3
4 data Boolean e = LitB Bool | OpB BEnum e e | If e e e
5 deriving Functor

```

Now, `Arith` and `Boolean` both have a continuation in the `e` parameter. We use these versions of the modules in the model (B.1.1, B.1.2). With a single parameter, these datatypes both fit the properties of *functor*: a type that can be mapped over with regard to some *value* parameter (Awodey, 2006). We will take advantage of that property later in the framework. The enums `ArEnum` and `BEnum` remain as specified earlier. However, if we try to compose these datatypes the same way we did previously, we quickly realise that this is not possible to achieve the recursion we want.

```

1 type Syntax e = Boolean e \ / Arith e

```

Unfolding `e` will give us `Syntax e` again; no reduction has occurred:

```

1 type Syntax e = (Boolean (Syntax e)) \ / (Arith (Syntax e))

```

The solution to this problem has two parts: first, we need to devise a way to take the sum of two *type constructors* instead of summing the *base types*, and second, we need to figure out how to inject a notion of termination into such constructed recursion.

Coproduct of Signatures

```

1 infixr 6 +
2 data (f + g) a = L (f a) | R (g a)
3 deriving Functor

```

The new sum operator, `+`, is also right-associative (we will find this property useful later on), and requires a bit more than syntactic sugar to work. This operator specifies that if two types have a single parameter `a`, or in other words are *functors* over `a`, and this parameter is the same for both types, we can *sum* the constructors alone; the same as with `Either`, only one of these constructors will actually be present at runtime, either in the `L` (left) constructor or `R` right constructor of the signature coproduct. The functors `f` and `g` will be now called subtypes of the coproduct `f + g`.

```

1 type Syntax = Arith + Boolean
2
3 symmetric :: Syntax (Syntax a)
4 symmetric = R $ If (R $ LitB True) (L $ LitAr 1) (L $ LitAr 2)
5
6 asymmetric :: Syntax (Syntax (Syntax a))
7 asymmetric = R $ If (R $ LitB True)
8   (R $ If (R $ LitB True) (L $ LitAr 1) (L $ LitAr 2))
9   (L $ LitAr 2)

```

With such defined coproduct we can construct small programs. `symmetric` gives the simple expression `if True then 2 else 3`, while `asymmetric` shows an example where the AST of the program is not symmetric: one of the children is a root to a tree with a greater height than the others. A graphical representation of the ASTs of these programs can be found in Fig. 2.2. We can see that we need to *unfold* the parameter `a` as many times as the longest path of the tree.

We can quickly see that while powerful, the current notation provides us with uncomfortable limits. First, the height of the program's AST needs explicitly given in the type signature. What is worse, since the leaf node does not have children, the type will always have an undefined trailing parameter.

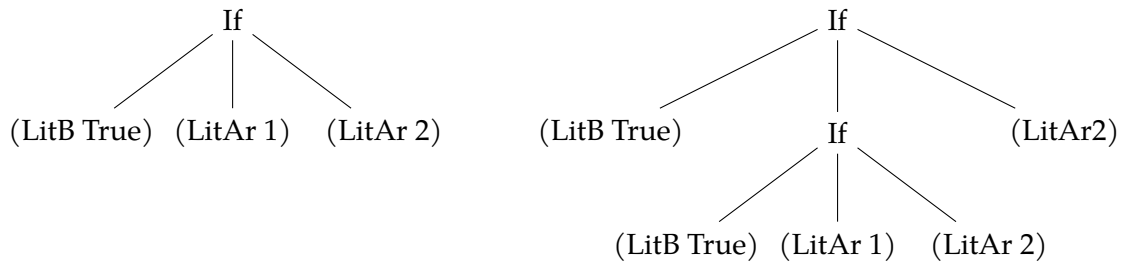


Figure 2.2: The symmetric(left) and asymmetric (right) ASTs

Fixed-point operator

```
1 data Fix f = In (f (Fix f))
```

To provide more flexibility in defining type signatures, we define the `Fix` operator. It hides the unfolding of type signature for us: `f` is present only once in the type signature of `Fix`, while the constructor of this datatype contains the top level occurrence of `f` and a nesting of it.

We can now rephrase one of our programs from the previous section:

```
1 symmetric' :: Fix Syntax
2 symmetric' = In $ R
3   $ If (In $ R $ LitB True) (In $ L $ LitAr 1) (In $ L $ LitAr 2)
```

Even though we fixed the complexity of the type signature of our programs, we introduced even more overhead in the program itself: now every actual syntax constructor has to be prepended by not one but *two* constructors creating unnecessary overhead. And recall that our coproduct `+` is right-associative: For a syntax composing of n modules, the right-most element will require $n+1$ constructors. This does not promise us usability we want to achieve.

Ideally, we would like to have a function with the signature:

```
1 inj :: f <: g => f a -> g a
```

The infix `<:` should be read as: `f` is a subtype of `g`. The function `inj` should be read as *type injection*: if `f` is a subtype of `g`, `inj` should allow to inject a variable of type `f a` into the type `g a` (a type sum including `f` in itself). The infrastructure necessary for this injection to work is given in the next subsection.

```
1 injF :: (f <: g) => f (Fix g) -> Fix g
2 injF = In . inj
3
4 symmetric'' :: Fix Syntax
5 symmetric'' = injF
6   $ If (injF $ LitB True) (injF $ LitAr 1) (injF $ LitAr 2)
```

We can now construct `injF`: injection into `Fix`. It allows to limit the number of constructors necessary for each AST node, which improves the readability of our programs, as can be seen in the third iteration of the program `symmetric`.

Two constructors per AST node sound manageable, but as our programs will grow, the code is bound to become less readable. We reached the limits of automation; to decrease the number of constructors we must encapsulate them within a separate function. We will call this kind of function with syntax limited to constructor calls and injection a *smart constructor*

```
1 true :: (Boolean <: f) => Fix f
```

```

2 true = injF $ LitB True
3
4 if' :: (Boolean <: g) => Fix g → Fix g → Fix g → Fix g
5 if' a b c = injF $ If a b c
6
7 bin :: (Arith <: g) => OpArith → Fix g → Fix g → Fix g
8 bin op left right = injF $ OpArith op left right
9
10 add :: (Arith <: f) => Fix f → Fix f → Fix f
11 add = bin Add
12
13 int :: (Arith <: f) => Int → Fix f
14 int = injF . LitAr

```

The presented *smart constructors* map to equivalent the syntax datatype constructors under `Fix`. Some constructors (for *false*, boolean binary operations and remaining arithmetic binary operations) can be easily derived based on the constructors present. Since boolean values are finite in count, we decide to split them into two constructors to further increase readability.

```

1 symmetric''' :: Fix Syntax
2 symmetric''' = if' true (int 1) (int 2)

```

Finally, we can define a very concise program `symmetric'''` with these smart constructors. We will use this setup to represent the syntax in tests in a readable manner.

Injection into the Functor Coproduct

In this section, we briefly explain the infrastructure for `inj` and the `<:` infix.

```

1 infix 4 <:
2 class (Functor f, Functor g) => f <: g where
3   inj  :: f k → g k
4   proj :: g k → Maybe (f k)

```

We can easily define our infix and the injection foreshadowed in the previous section. We also define `proj`: a transformation in the opposite direction, that projects the coproduct to the expected value *if that value is present at runtime*. To handle this uncertainty, the projected value is wrapped in the **Maybe** monad. The only thing left to do is to allow Haskell to automatically infer the subtype relation for us.

```

1 instance Functor f => f <: f
2 instance (Functor f, Functor g) => f <: f + g
3 instance (Functor f, Functor g, Functor g', f <: g') => f <: g + g'

```

With these three instances, implementation of which can be found in the source code, any functor can be freely injected. The first base case instance claims that a functor is always a subtype of itself. The second instance, also a base case, promises that if the functor is the leftmost summand, it is also always a subtype of the sum. Finally, with the third, recursive instance, we can say that if a functor is a subtype of the right summand of the sum, it is also a subtype for the full sum.

This framework is not as flexible as one can imagine; for example, while a supertype can be arbitrarily nested, the subtype always must be a concrete functor, as the instance rules do not allow for it. This limitation will come back in later elements of the pipeline.

2.2.3 Syntax

We can summarise the takeaways from this section.

Each of the modules of the language has a Syntax component that represents its fragment of WebDSL. This component is a functor datatype, and the complete WebDSL syntax is a functor sum of all of the syntax datatypes.

2.3 Modular Semantics

In this section, we will present the IR used for representing the semantics and effects with the data defined in the Effects component presented in Fig. 2.1, and explain how we compose the values and effects to embed them easily in the IR.

```
1 data Free f a = Pure a | Op (f (Free f a))
```

The Free monad is a carrier of the IR. It has two parameters: *a*, the pure value that is the outcome of the computation, and *f*, which carries an operation to be applied that might generate a side effect. We point that in the *Op* constructor, *f* is a functor over the Free monad itself (Szamotulski, 2018; Awodey, 2006). The monad itself represents a tree structure: the *Pure* constructor defines a leaf of the tree containing only the pure value, and the *Op* constructor defines a node in the tree, where the amount of children of that node is defined by the functor datatype *f*.

Consider the program *symmetric* introduced in the previous section. The *Arith* datatype doesn't raise any effects, and the *Boolean* datatype raises one effect:

```
1 data Cond k = Cond Bool k k
2 deriving Functor
```

The *Cond* datatype (A.1.1) represents conditional operation such as one in the *if/then/else* operator, with a single constructor: it holds a decider value and two continuations, which are other nodes of the Free monad. One might ask why we consider *conditional operation* to be an effect, when Haskell provides an implementation for this construct; the answer being, we want to limit the Haskell syntax used in the denotation. More detailed answer to this question will be given in Section 2.4.

```
1 symmetricFree :: Free Cond Int
2 symmetricFree = Op (Cond (True) (Pure 1) (Pure 2))
```

The *symmetricFree* variable represents a possible Free monad representation of the program denotation. We can immediately notice two drawbacks of this situation: This type definition of the IR representation only allows for a single type of effect to occur in scope and a single type of value to be the outcome. This is not sufficient; we need both a way to compose the values and the effects.

2.3.1 Free Monad: Value Composition

In this section, we present how we define the set of values allowed to occur as the *Pure* leaf value of the Free monad.

```
1 type IR = Free Cond _
2
3 reverseIf :: IR
4 reverseIf = (Op (Cond (True) (Pure False) (Pure True)))
5
6 symmetricFree' :: IR
7 symmetricFree' = reverseIf >>= (bool → Op (Cond (bool) (Pure 1) (Pure 2)))
```


In the code snippet above we see the highlight of the problem: We would like to have a single type defining all allowed values. We could use the type `\/` for value coproduct defined in the previous section:

```
1 type IR = Free Cond (Bool \/ Int)
2
3 reverseIf' = (Op (Cond (True) (Pure (Left (False))) (Pure (Left True))))
4 symmetricFree' = reverseIf
5   >>= (\bool → Op (Cond (bool) (Pure (Right 1)) (Pure (Right 2))))
```

This solution works only for as long as we do not take into account recursive values in the language, for example lists.

```
1 data Col e = LitC [e] | OpIn e e
2   deriving Functor
```

Consider another syntax module to give list operations presented above, `Col` (B.1.5). It has two constructors, `LitC` to represent list literals and `OpIn` to represent the contains or `elem` operation. With this module in scope, we would like to be able to also have arbitrarily nested lists as the output of the program.

We can take advantage of the fact that lists are also a functor; if we raise the terminal values to terminal functors, we can reuse the infrastructure from the previous section.

```
1 data Lit v e = V v deriving Functor
2 type IR = Free Cond (Fix (Lit Int + Lit Bool + []))
3
4 reverseIf'' = (Op (Cond (True) (Pure (In (R (L (False)))))) (Pure (In (R
5   (L (False))))))
6 symmetricFree'' = reverseIf
7   >>= (\bool → Op (Cond (bool) (Pure (In (L 1))) (Pure (In (L 2)))))
```

We define the `Lit v e` datatype to raise any terminal value `v` to a functor. `Lit` is a general datatype that allows to limit the infrastructure in case each of the types was raised to functor separately, and allows to use common smart constructors throughout the framework. Further, we use the functor sum type `(+)` to define the coproduct of the allowed values, and enclose them in the `Fix` datatype to allow arbitrary recursion. With such defined type, we have the freedom to define IRs with a wide range of resulting values.

It is important to notice that this type definition forcing so many nested constructors on any instance of `Pure` might be considered difficult to read - it is important to mentioned here that the IR is not meant to be witnessed by the user: it is generated by the first mapping and deconstructed by the second, as seen in Fig. 2.1. It carries the denotation, but its structure is not necessary to be read by the user of these denotation semantics. Instead, it must be generic enough to allow simple infrastructure for both of the mapping steps.

2.3.2 Free Monad: Effect Composition

We ended up with a robust way to define result values, but we still need to consider a way to represent the Intermediate Representation type that allows apply more than one type of effect.

```
1 data Abort v k = Abort v
2   deriving Functor
```

Consider the `Abort` effect (A.1.2): It typically represents halting the computation. It takes a value `v` that will provide the result in place of the halted computation. We use this effect to represent the `return` statement that typically occurs in a function.

Since both `Cond` and `Abort` effects are functors over the IR continuation, we can simply use the functor sum to combine them.

```

1  type Value = (Fix (Lit Int + Lit Bool + []))
2  type Eff = Cond + Abort Value
3  type IR = Free Eff Value
4
5  reverseIf :: IR
6  reverseIf = (Op (L (Cond (True) (Pure (In (R (L (False)))))) (Pure (In (R
      (L (False)))))))

```

We have defined a general way for combining allowed result values and effects in the scope for the IR that gives us the desired modularity.

2.4 Denotational Mapping

In this section, we will present the infrastructure we use for the mapping between the syntax definition and the IR. This mapping presents the core of the denotational semantics we strive to give in this work. The modularity allows us to extend the set of covered syntax without collisions between different modules. First, we will give the infrastructure for the modular definition. In Section 2.4.1 we describe the steps necessary to apply the mapping to the syntax described in Section 2.2, followed by extending the presented denotational mapping to include environment in section 2.4.2. Finally, we present the rules we use to limit the syntax of the denotational semantics in Section 2.4.3.

```

1  denoteArith :: Arith (Free f v) → Free f v
2  denoteBool  :: Boolean (Free f v) → Free f v

```

To achieve this modularity, for every syntax datatype we require a mapping function from a syntax AST node to the IR tree representation. This function will have the same general signature for any syntax element in scope. The functions `denoteArith` and `denoteBool` presented above provides us with that exact mapping. These functions are the algebras: they determine how the different constructors of a datatype affect the final outcome, specifying one step of recursion that turns the syntax sum into the semantic representation of the program.

```

1  class Functor sym ⇒ Denote sym eff v where
2    denote :: sym (Free eff v) → Free eff v

```

We can generalize that function to a type class `Denote`. It is parametrized by the `sym` (symbol), which stands for the particular datatype, `eff` (effects) and `v` (end values) parameters.

```

1  instance (Denote sym1 eff v, Denote sym2 eff v) ⇒ Denote (sym1 + sym2)
      eff v where
2
3    denote s = case s of
4      (L f) → denote f
5      (R f) → denote f

```

For any arbitrary ordering of syntax symbols, we can define a general composition of their denotations, provided they share the same effects and values. In the instance defined above, as long as both symbol summands have an instance of `Denote` in scope, the `denote` function should pass the call to the subtype available at runtime. While this approach lends the modularity for syntax components, its strong limitation is that every instance of the Denotation

class in scope needs to share the same *effect* and *value* parameters in order to call the composition instance.

To overcome this limitation, for every syntax datatype we define a function *without instantiating them* as members of the `Denote` class. Instead, we make use of the type class defined by `<`: infix to give constraints on the `eff` and `v` parameters of the type signature.

```
1 denoteBoolean :: (Cond <: eff, Lit Bool <: v) => Boolean (Free eff (Fix v)) -> Free eff (Fix v)
2
3 denoteArith :: (Functor eff, Lit Int <: v) => Arith (Free eff (Fix v)) -> Free eff (Fix v)
```

In the code snippet above, we give an example of type signatures we need for denoting the `Arith` and `Boolean` syntax datatypes that conform to this notation. In order to define constraints on the *value* parameter of the `Free` monad, we need to specify that it is embedded in the `Fix` datatype, however for denotation functions that do not require constraints on values, this specification will not be necessary, so the type class `Denote` also does not have to hold this specification.

Instead, we will need to instantiate the type class `Denote` for a concrete effect type coproduct and a concrete value type coproduct for every new composition of syntax.

```
1 type Symbols = Arith + Boolean
2 type V = (Fix (Lit Int + Lit Bool))
3 type Eff = Cond + Abort V
4
5 instance Denote Arith Eff V where
6   denote = denoteArith
7
8 instance Denote Boolean Eff V where
9   denote = denoteBoolean
```

In the code snippet above, we give an example of that instantiation for the small language subset we discussed in the previous sections. Assuming the functions `denoteArith` and `denoteBoolean` with the signatures as defined before, we can define the instantiation.

This approach has a downside: it introduces some infrastructure overhead proportional to the number of separate syntax datatypes in scope. We will discuss in the following chapter the methods we use to manage this overhead.

2.4.1 Applying the denotation

In this section, we will explain the last step necessary to apply this `denote` function to the syntax structure presented in Section 2.2 in order to obtain the IR presented in section 2.3. Consider the program `symmetric`, as presented in Sections 2.2 and 2.3:

```
1 symmetricSyntax :: Fix Symbols
2 symmetricSyntax = In $ R $ If (In $ R $ LitB True) (In $ L $ LitAr 1) (
3   In $ L $ LitAr 2)
4
5 symmetricIr :: Free Eff V
6 symmetricIr = (Op (L (Cond (True) (Pure (In (R 1))) (Pure (In (R 2))))))
```

These program representations make use of type aliases defined in the section before. In order to map from one tree to another, we need to apply the `denote` function recursively.

```
1 foldD :: Denote f eff v => Fix f -> Free eff v
2 foldD (In f) = denote $ fmap foldD f
```

We can achieve this with the function `foldD` (fold denote) defined in the code snippet above. Fold is the core element of this part of the framework, applying the *algebras* to transform syntax trees into semantic trees. Fold *unpacks* the syntax from the `Fix's In` constructor and, taking advantage of the fact that every syntax datatype is a functor, recursively applies `foldD` with `fmap` (the functor mapping function).

```
1 mappingEquivalence :: Bool
2 mappingEquivalence = (foldD symmetricSyntax) == symmetricIr
```

The `mappingEquivalence` equation compares the application of `foldD` to previously defined `symmetricSyntax`, with `symmetricIr` holding the IR of the program. Assuming all constructs that are embedded in the IR type have instances of the `Eq` type class, the `mappingEquivalence` would evaluate to `True`.

2.4.2 Environment Extension

In this section, we will explain some practical aspects of the denotational mapping that needs to be addressed: how to include an environment? The current formulation is not sufficient. We will explain the possible solutions and the problems they raise on the example of functions syntax module.

```
1 type FunName = String
2 data Fun e = Return e | FCall FunName [e] deriving Functor
```

Consider the `Fun` (function) datatype. It specifies two operations: `Return` - returning the value in `e` from the function, and `FCall`, which defines the call of function with the name defined by `FunName` and arguments to the function in `[e]`. Functions are also being added to the scope in a different processing step that needs to be accommodated; More details about it can be found in Section 3.1.

This shows a gap in our current framework definition: in order to dereference the `FunName`, we need to do a *lookup* into some environment. This environment *could* be, perhaps, defined by an effect: an ML-style `State` definition that holds the current scope in some collection of key-value pairs (Staton, 2010). We chose this effect as it provides the operations necessary to interface with the environment.

```
1 data MLState m v k = Ref v (m → k) | Deref m (v → k) | Assign (m, v) k
```

The code snippet above presents an example of such ML-style `State` definitions A.1.3. It is defined by three parameters: `m`, the key of the tuple, `v`, the value of the tuple, and `k`, the continuation computation. It also has three constructors: first of them is `Ref`, which accepts a value and *feeds* into the continuation a key by which this value is reachable. Second constructor, `Deref` defines an operation that accepts a *key* and passes to the continuation a value associated with it. Finally, `Assign` accept a tuple of key and value and doesn't return any information to the remaining computation.

This approach has two downsides: first, it decouples the store from the syntax tree structure. The denotational semantics don't strictly require for the environment to contain *only* the variables that should be in scope in the given moment; the program has passed static verification, so it does not try to refer to variables that are out of scope.

However, having a faithful scope aids with correctly solving *name shadowing* that is still a practical concern for us due to the *applying implementation* step. There are perhaps ways to overcome it; an alternative would be to expect the static verification to alter all names in scope to disambiguate name references. It is possible, but it might cause names in scope to be less readable for users. Alternatively, the denotation function could explicitly execute not only extending the scope with new values, but also emptying it; this solution introduces additional pollution of the denotation that takes away from the clarity of the code.

The second problem is much more severe. Function definitions are pieces of *unexecuted* code. Ideally, they would be an instance of the free monad, with the same value of the `eff` parameter. This raises an issue that at first glance is not obvious. Consider an example signature of the `denoteFun` function:

```
1 denoteFun :: (Abort v <: eff, MLState FunName (Free eff v) <: eff) =>
  Fun (Free eff v) -> Free eff v
```

Constraints in this signature put two effects in scope: the previously discussed `Abort v`, and the probable `MLState` for key-value pairs of `FunName` and the `Free` monad, that is supposed to provide is with the scope of functions. Looking at this definition, perhaps the issue is still not obvious. However, when we try to define the type of `IR` of the language subset with this particular effect:

```
1 type Eff = Abort V + MLState (Free Eff V) + Cond
2 type IR = Free Eff V
```

As earlier, the type `V` holds the values in scope and `Eff` defines the effects. At this point it has become apparent: *type aliases* cannot be recursive; only datatypes can be. The reason for that is that a type alias does not actually create any new data, it only aids the programmer by providing a *synonym* to the given type. If that phrase contains itself, the type cannot resolve. Mutually recursive type aliases are not allowed either.

There is perhaps a way to overcome this: instead of an `IR` version of the function, the effect could hold the *syntax* representation of the function's body. This would allow us to store the function in the effect row:

```
1 type Eff = Abort V + MLState (Fix Symbols) + Cond
```

While technically feasible, it would require from the denotation to explicitly call the recursive folding `foldD` function every time a function definition is dereferenced. This solution has the same downside of polluting the denotational mapping function with unnecessary function calls.

With all these obstacles and imperfect solutions discussed, we propose an alternative approach: explicitly passing an *environment record datatype* through the `denote` function.

```
1 data Env eff v = Env { functionDefs :: [Function eff v] }
```

The `Env` record representing the environment is parametrised by the effects and values of the free monad. We present a simple version with a single field, but this definition of environment allows us to easily extend it with other fields when needed, without losing backwards compatibility with earlier defined modules. We can now redefine the `Denote` type class:

```
1 type EnvFree eff v = Env eff v -> Free eff v
2
3 class Functor sym => Denote sym eff v where
4   denote :: sym (EnvFree eff v) -> EnvFree eff v
```

Since in most cases we will require the same `eff` and `v` definitions from the environment and the free monad types, we define the alias `EnvFree` to aid conciseness of further code snippets. The `denote` function is formalized as follows: the children of the currently denoted node in the program AST require the *environment* as input and output the free monad. The `denote` function accepts an instance of this environment and results in the free monad that holds the semantics of the processed AST node. This definition will be the basis of our framework, with some modifications for different components of `WebDSL`.

2.4.3 Formal Requirements of the denote Function

We have defined the types which should hold for both ASTs of the program. Now, we will give (partial) implementations of the denoting functions necessary to apply this mapping; as we want the mapping to *represent* the denotational semantics, we will also describe the formal requirements that the code of the mapping functions should conform to in order to consider it denotational semantics. We define these rules and limit the allowed syntax within the denotational mapping to make a distinction between these semantics and an arbitrary Haskell program.

```
1 denoteArith :: (Functor eff, Lit Int <: v) => Arith (Free eff (Fix v))
   → Free eff (Fix v)
2 denote (LitAr int) = return $ injF $ V int
```

In the code snippet above, we give the denotation of the arithmetic literal, integer. This is the leaf of the AST mapped to the leaf on IR, not presenting yet any dynamic semantics. This implementation makes fault of exposing the infrastructure elements such as Fix injection injF. To avoid that, we present first principle of the denotational mapping:

when applicable, use *smart constructors* (Section 2.2.2) (2.1)

This includes constructing both leaf values and operations of the free monad.

```
1 v = Pure . injF . V
2
3 denoteArith (LitAr int) = v int
```

We define `v`, the smart constructor of pure values. It allows us to hide the infrastructure of the IR while emphasizing the operations, or lack thereof, that are executed for the given syntax node.

```
1 unV :: Lit a e → a
2 unV (V a) = a
3
4 projV :: (Lit a <: g) => Fix g → Maybe a
5 projV (In fix) = unV <$> proj fix
6
7 op :: (Functor eff, Lit Int <: v) => (Int → Int → Int) → Maybe Int →
   Maybe Int → Free f (Fix v)
8 op operand (Just e1) (Just e2) = v $ operand e1' e2'
9
10 denoteArith (OpArith Adda b) env = do
11   a' ← a env
12   b' ← b env
13   op (+) (projV a') (projV b')
```

In the second code snippet, we give a typical binary operation denotation. First, we present `unV`, which unpacks the literal value from the functor lift, and `projV`, which maps `unV` on the coproduct projection. Second, we present the smart constructor `op` for (arithmetic) binary operations. Since we operate under the assumption that the program passed static verification, we can assume that both values passed to the equation will have the correct type - `Int`. However, the `proj` function always returns the type embedded in the `Maybe` monad. Therefore, in all cases like this one we can safely omit any case where either of the values passed evaluates to `Nothing`. Finally, we give the denotation of the addition operation. This raises more rules that we want to adhere to:

define the denotation with the `do` notation (2.2)

within a **do** block, limit syntax to *smart constructor* calls and monadic operations (2.3)

allow pattern matching only by function calls (2.4)

define the variable names in an easily identifiable manner (2.5)

In rule 2.4 we specify that pattern matching on values should not be done with inline **case ... of** syntax, and instead only realized by pattern matching in function arguments, as it is done in the `op` function.

```

1  cond :: (Cond <: f) => Maybe Bool -> Free f fix -> Free f fix -> Free f
   fix
2  cond (Just bool) k1 k2 = Op . inj $ Cond bool k1 k2
3
4  denoteBool (If c a b) env = do
5  c' <- c env
6  cond (projV c') (a env) (b env)

```

In the last code snippet, we give example of *smart constructor* encapsulating an effect.

The notion of *smart constructors* might seem not concrete enough. Their primary purpose is to hide the infrastructure invocation when constructing the `Free` monad. However, in some cases there are operations such as `op` for binary operations that do not emit any effects, and we do not expect any benefit from being able to easily change the runtime implementation of that effect with a handler. In that case, the operation is kept *pure* within the smart constructor.

In some cases, we break these rules. Some denotation definitions benefit from encapsulating a reusable function that also conforms to the denotation rules we gave; this function call is distinct from smart constructors. When interacting with environment, we apply handlers *inline* in order to centralize handling different aspects of environment. Be *inline*, we mean within the denotational mapping and not in the *applying handlers* step of the pipeline in Fig. 2.1. This allows us to construct an effect and immediately deconstruct it, without mentioning it in the effect row, which is sometimes impossible (e.g. as discussed in Section 2.4.2). Using a handler *inline* highlights the effectful nature of the environment and integrates the style with the rest of the codebase. It also suggests we might benefit from using *higher order effects* (Poulsen and Rest, 2023) in our framework; this will be discussed further in Section 4.3.

2.5 Handling Effects

In this section, we will explain how we use handlers to provide implementation for the semantics in IR. Where effects are supposed to be broad and general to allow defining laws and theories about them, handlers are supposed to be more concrete providers of *implementation* for the semantics described with the effects, fit for the particular use case. With the division of effects and handlers, it should be easy to replace one handler with another, changing the *computed output* of running the semantics. We hope to give the reader intuition on how to read and construct handlers to achieve the desired outcome.

Consider the language subset and example program from Section 2.3.

```

1  type IR = Free Cond V
2
3  symmetric :: IR
4  symmetric = Op (Cond (True) (Pure (In (L 1))) (Pure (In (L 2))))

```

A basic handler type is defined as follows:

```

1  data Handler f a f' b = Handler
2  { ret  :: a -> Free f' b
3  , hdlr :: f (Free f' b) -> Free f' b }

```


It is defined by a record datatype with two fields; the `ret` field determines the operation on *leaf* value, and `hdlr` defines the operation on node elements of the free monad IR. Notice that it is characterized by *two* effect functors: `f` and `f'`. We give meaning to these functors in the `handle` operation:

```

1  handle :: (Functor f, Functor f') => Handler f a f' b -> Free (f + f') a
   -> Free f' b
2  handle h (Pure a) = ret h a
3  handle h (Op f) = case fmap (handle h) f of
4  L y -> hdlr h y
5  R y -> Op y

```

The function takes a `Handler` variable and a free monad, outputting the monad modified. The signature of `handle` defines the relation between these two functors. First functor, `f`, must describe a single effect, which follows from the *right-associativity* of the `(+)` coproduct sum. The second functor can be any functor coproduct or a single functor *without* the `f` functor. The `handle` function transforms the free monad such that it *removes* the rightmost effect of the effect row.

The implementation of `handle` applies the functions defined in the `Handler` variable to the free monad. In case of the `Op` node of the monad, the function recursively applies the itself and maps on the functor coproduct `f + f'`. In case it is the `L` constructor, it simply applies the function in `hdlr` field of the `Handler`. Otherwise, it reconstructs the encountered effect coproduct while removing the `f` functor from the effect row.

These semantics define a general mapping of the free monad; we can generalise the monad traversal into a separate function to emphasize the handling semantics.

```

1  fold :: Functor f => (a -> b) -> (f b -> b) -> Free f a -> b
2  fold gen _ (Pure x) = gen x
3  fold gen alg (Op f) = alg (fmap (fold gen alg) f)

```

The `fold` function takes two functions as input: a pure value map and an operation map, and recursively maps them over the tree structure defined by the monad. We can use this function to redefine `handle`:

```

1  handle h = fold (ret h) (\x -> case x of
2    L y -> hdlr h y
3    R y -> Op y)

```

With this infrastructure, we can define a handler to *run* the symmetric program.

```

1  condition :: (Functor g) => Handler Cond a g a
2  condition = Handler
3    { ret = pure
4      , hdlr = \(Cond v thenC elseC) ->
5        if v then thenC else elseC }

```

The `Cond` handler `condition` is very straightforward. It maps the effect to the *if/then/else* Haskell syntax, choosing between the two continuations. We can now attempt to apply this handler.

```

1  run :: V
2  run = case handle condition symmetric of
3    (Pure a) -> a

```

This `run` function applies a single handler to the IR defined in `symmetric`. Since the type of IR has only one effect in scope, this is sufficient to give certainty that the value of IR is only the `Pure` constructor. However, Haskell allows us to pattern match on any state of the free monad without raising an error; we would like more security for the framework.

```
1 type End k = deriving Functor
```

To achieve this security, we define the End functor; its purpose is to flag the end of the effect row. This functor does not have any constructors, giving a guarantee that and IR of type `Free End v` will at runtime be the Pure constructor.

```
1 unwrap :: Free End a → a
2 unwrap (Pure x) = x
3 unwrap (Op f) = case f of
```

Defining a Handler for End would allow to place the effect in anywhere in the effect row; instead, we define the unwrap function, which *unpacks* the value from the Pure constructor. We can now redefine the IR for symmetric:

```
1 type IR = Free (Cond + End) V
2
3 run' :: V
4 run' = unwrap $ handle condition symmetric
```

The second iteration of run function `run'` showcases an application of the End wrap and secure unwrap interface. This concludes the basic principles of providing the implementation for a free monad program representation with the use of handlers. In the following subsections, we will discuss two necessary extensions to the handling semantics.

2.5.1 Handling with State

Let us consider the Cond effect again. Assume we would like to debug a program embedded in the free monad; for example, we want to inspect the values that are passed to Cond that define the chosen branch. To do that, we need a notion of *state*. The current Handler definition does not support that. In this section, we will present an alternative Handler formulation that allows working with state.

```
1 data Handler_ f a p f' b = Handler_
2   { ret_  :: a → (p → Free f' b)
3     , hdlr_ :: f' (p → Free f' b) → (p → Free f' b) }
```

The second iteration of handler record `Handler_` provides us with the notion of state. It is built similarly to `Handler` with a single difference of the `p` parameter. The structure of `ret_` and `hdlr_`, the two fields of this record, requires `p` to be provided before a free monad output can be given. We can use it to provide the debug Cond handler.

```
1 condition' :: (Functor g) ⇒ Handler_ Cond a [Bool] v g (a, [Bool])
2 condition' = Handler_
3   { ret_  = curry pure
4     , hdlr_ = \(Cond v thenC elseC) list →
5       let list' = v : list
6         in if v then thenC list' else elseC list' }
```

In the handler `condition'`, the `ret_` function carries the two input variables into the expected tuple output, and the `hdlr_` function prepends the value received in `v` argument to the state parameter of the handler, and passes the new variable `list'` to both of the continuations.

Before we can apply this handler, we need to also define the new handling function.

```
1 handle_ :: (Functor f, Functor f') ⇒ Handler_ f a p f' b → param →
2   Free (f + f') a → Free f' b
3 handle_ h param value = fold (ret_ handler)
```



```

4  (\case
5    L f → hdlr_ h f
6    R f → \param → Op (fmap (\apply → apply param) f))
7  value param

```

We build the `handle_` function with the previously described `fold` function. The base case of the fold doesn't note any differences from the stateless `handle` function: it simply applies the base case from the handler record. The only difference is in the R case of the recursive component of fold: the function maps the parameter through the monad's tree structure in order to extract the monad value out of the function requiring `param` as input.

Now, we can make use of the new handler:

```

1  run'' :: (V, [Bool])
2  run'' = unwrap $ handle_ condition [] symmetric

```

In the second iteration of `run`, we use the same effect constructor `Cond` as earlier, this time with a different handler. The `run` function returns two values, the pure computation result `V` and the side effect description that we needed for debugging. The `handle_` function requires and initialization of the state parameter. In this case we provide an empty list, but it also can be some notion of populated state, as we would typically need for representing the `Reader` effect.

2.5.2 Handling with IO Operations

So far, we have presented the infrastructure for executable reference semantics. However, for some use cases, we want the semantics to act as an executable prototype with easily replaceable implementations. This raises some additional requirements for the handlers. For example, in order to validate that our semantics are consistent with the `WebDSL` specification, we need to some handlers to perform IO operations. In this section, we will explain different variants of including IO operations in handlers, which in some cases require a different formulation of both (stateless) `Handler` and `handle` constructs, as well as the infrastructure necessary for these constructs.

If the IO effect occurs only in the return function of the handler, we can reuse the already defined handler records to implement such a handler. Consider this *Writer* effect and its handler:

```

1  data Writer k = Write String k deriving Functor
2
3  writeToFile :: String → String → IO ()
4
5  writerIoH :: (Functor g) ⇒ String → Handler_ Writer a [String] g (IO a
6  )
7  writerIoH fileName = Handler_
8    { ret_ = \v store → do
9      writeToFile store fileName
10     return $ pure v
    , hdlr_ = \(Write s k) store → k $ s : store }

```

The datatype `Writer` has a single constructor `Write`, which takes arguments of type `String` for simplicity. On line 3 of the code snippet, we define the type of some arbitrary `writeToFile` function that writes the data (first argument) to some file specified by its path (second argument), the possible implementation of which is left to the discretion of the reader. The function, to work as expected, must result in the IO monad.

Finally, we give the handler of this effect, `writerIoH`. this handlers takes as input the file path of the target of the data. In the `hdlr_` case, it merely prepends the state parameter with

received values. In the `ret_` case, when all the data to write has been collected, we invoke the `writeToFile` function and pass the IO monad inside the free monad.

```
1 runWriter :: Free (Writer + End) v → IO v
2 runWriter e = unwrap $ handle writerIoH e
```

The function `runWriter` gives possible semantics of applying the `writerIoH` handler, lifting IO outside of `free`.

Immediate IO in Handling Case

The alternative generation of IO immediately on handling a constructed effect raises more issues. We will continue with using the `Writer` effect in our example. Perhaps we have multiple effects in scope writing to the same file, so we require immediate dumping of any writtern value. We might want to have the following function in the `hdlr` field of `Handler`:

```
1 hdlrCandidate :: String → f (Free f' b) → IO (Free f' b)
2 hdlrCandidate fileName (Write s k) = do
3   writeTofile s fileName
4   return k
```

The function `hdlrCandidate` takes as input a `String` representing path to target file and the `Writer` effect instance. It executes the IO actions with `writeToFile` and attempt to return the computation continuation. However, even if we assume that the input free monad we receive is not embedded in IO, the embedding of the output monad will not match the `Handler` signature:

```
1 hdlr :: f (Free f' b) → Free f' b
```

Likewise, it will not conform to the `hdlr_` type singature. Instead, we need to define new handler records that will allow us arbitrarily emitting IO operations.

```
1 data IOHandler f a f' b = IOHandler
2   { ioRet  :: a → IO (Free f' b)
3   , ioHdlr :: f (IO (Free f' b)) → IO (Free f' b) }
```

The record `IOHandler` provides us with signatures that permit embedding the monad in IO operations. We can use this record type to define handling with the implementation we wanted to use for `hdlrCandidate`.

```
1 writeH :: Functor eff' ⇒ String → IOHandler Writer v f' v
2 writeH fileName = IOHandler
3   { ioRet  = pure
4   , ioHdlr = \(Write s k) → do
5     writeTofile s fileName
6     return k }
```

Before we can attemple to include this handler in a `run` instance, we have to define the function applying it, equivalent to `handle`.

```
1 ioHandle :: ( Functor f, Functor f' ) ⇒ IOHandler f a f' b → Free (f +
2   f') a → IO (Free f' res)
3 ioHandle h = fold (ioRet handler) (\ x → case x of
4   L y → ioHdlr handler y
5   R y → _ )
```

The `IOHandle` functions takes as arguments the previously defined `IOHandler`, and a free monad without any `IO` operations, and while handling the `f` effect, embeds the free monad in IO. The base case and effect handling case are as before, but we notice that it is impossible

to define the case mapping over f' : Monads are by nature *not* distributive over each other, so this needs to be solved on a case-by-case basis.

```

1  class Distributive m f g where
2    distr :: f (m (Free g a)) → m (Free g a)
3
4  instance (End <: g) ⇒ Distributive End IO g where
5    distr e = case e of

```

We define the type class `Distributive` with three parameters: m , the monad we want to distribute over the free monad, f , the functor representing the handled effect, and g , which in the context of `ioHandle` will represent f' , with f giving the rightmost functor in f' co-product sum. This definition of distributive monad is not the standard formulation, but this rephrasing allows us to define distribution of a concrete effect f over $(m (Free g a))$ under the condition that f is a subtype of g ($f <: g$).

We also define the instance of this type class for the flag functor `End`. Since it has no constructors, the implementation is trivial.

```

1  ioHandle :: (Distributive f' IO f', Functor f, Functor f') ⇒ _
2  ioHandle h = fold (ioRet handler) (\x → case x of
3    L y → ioHdlr handler y
4    R y → distr y )

```

With the aid of the newly defined type class, we can complete the `ioHandle` implementation. In the mapping over f' , we simply apply the `distr` function of the `Distributive` type class.

```

1  runWriter' :: Free (Writer + End) v → IO v
2  runWriter' e = do
3    action ← ioHandle writerH e
4    return $ unwrap action

```

We can now define the function that applies the `ioHandler` in its runtime.

Chapter 3

WebDSL components

In the previous chapter, we introduced the basic principles of the framework we are using and the background behind it. The framework is well suited to represent typical features of any general purpose programming language. However, WebDSL is more than that, and some components require different adaptations or even extensions to the base framework. This chapter will describe the different WebDSL components and the necessary adjustments to the framework.

First, we will present the definitions component and the pre-processing step they raise in Section 3.1. It is followed by global variables, which insert another step into the processing pipeline, separate from the previously mentioned definitions in Section 3.2. In Section 3.3 we present the templates component, which is the *top layer* of WebDSL programs, and an extension to the framework for multi-sort syntax that is necessary for that component to have its semantics properly described. Finally, we present the multi-phase processing mode which works by running the same syntax multiple times, each with different semantics.

3.1 Definitions

In Chapter 2.4.2, we introduced functions component submodule that gives function call syntax. In that chapter, we focus on *retrieving* the function from the environment. However, before a function can be called, it has to be *described* outside the current execution AST. In this section, we will describe the top-level infrastructure we use to store and process the definitions and the entry point to the program.

In WebDSL, functions are one of many definitions that appear exclusively on the top level of a program - they are never arbitrarily nested in the program AST or anonymous.

```
1 function callee(a : Int) : Int { return (a * 5 + 2); }
2 function caller() : Bool { return (callee(4) > 10); }
```

In the code snippet above, we give two functions, `caller` and `callee`, which contain some basic language expressions. This WebDSL program does not have an entry point - it is created by a different component of the language. However, every WebDSL program consists of a list of definitions like the ones presented above; we need to find a way to *preprocess* these definitions so that they can be found in the environment at the entry point.

```
1 type FunName = String
2 type ArgName = String
3 data FDecl e = FDecl FunName [ArgName] e e deriving Functor
4
5 fDecl :: (FDecl <: s) => FunName -> [ArgName] -> Fix s -> Fix s -> Fix s
6 fDecl fName args body cont = injF $ FDecl fName args body cont
```

```

7
8 program = fDecl "callee" ["a"] (...)
9   $ fDecl "caller" [] (...)
10  $ entryPoint

```

If we wanted to reuse the current denote definition, it is possible with some work: we would have to give a less faithful program syntax where the definitions structurally wrap the program entry point. In the code snippet above, we give an example of that syntax. The last argument of the `FDecl` (function declaration) datatype is the program continuation, which is the only way we can pass the altered environment to the remainder of the program. In the example syntax program, we give an example of how that structure would look like.

While this approach *technically works*, it costs us some modularity. First, we limit the entry point to single occurrence – nested in all the function definitions. In reality, a program describes a web server it generates, and after being initialized it accepts HTTP requests as entry points and generates responses based on that. Even though it is out of scope of the functions component, it is an important feature to consider when thinking about modularity of the syntax. Second, the `FDecl` datatype is used not only as part of the syntax, but also carrying the function definition as part of the environment. Having it store the program continuation as one of the values will require additional operations of removing the unnecessary continuation from the definition, and the result will still have the redundant variable in its definition. Finally, this limits our capabilities in case other steps need to be added between processing the program definitions and its entry point - which is a case that will be discussed in Section 3.2. In this case, the main limitation stems from the fact that the discussed formulation requires the same effect row for both definitions and the entry point. If we want different effects for the intermediate step between definitions and the entry point, it is not possible with the nesting approach.

Instead, we propose to structurally separate the definitions from the entry point in the framework, and process them in separate steps. In Figure 3.1 we present the pipeline that we will construct in the remaining part of this section.

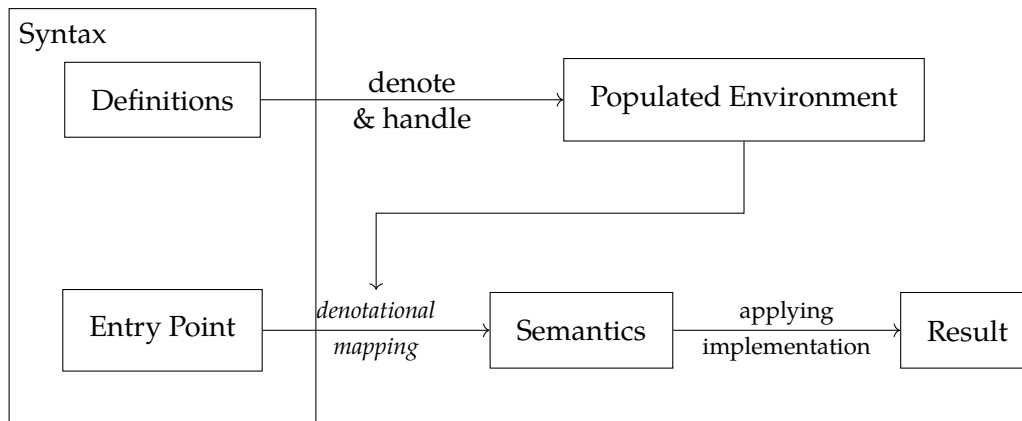


Figure 3.1: Pipeline including definitions

```

1 data FDecl e = FDecl FunName [ArgName] e deriving Functor
2
3 denoteDef :: (MLState FunName (Function eff v) <: eff') => FDecl (
4   EnvFree eff v) -> Free eff' ()
5 denoteDef decl@(FDecl name _ _) = do
6   (name :: FunName) <- ref decl
7   return ()

```

In the code snippet above, we present the denotation of function declarations. The `FDecl`'s parameter is `EnvFree eff v` (introduced in 2.4.2), which is the same as the parameter of the entry point's syntax. However, it does not take the environment as input and output `Free` with different parameters. This is because the environment is populated via effects. Due to limitations on Haskell's type signatures discussed in Section 2.4.2, this is possible only if the effect representing the environment (in this case `MLState FunName (Function eff v)`) is not parametrized by the same effect row it is a part of. Since we interact with the environment exclusively via handlers, moving that interaction to the handling step allows us to give less convoluted denotation.

```

1  class (Functor eff', Functor sym) => DenoteDef sym e eff' where
2    denoteDef :: sym e -> Free eff' ()
3
4  instance (DenoteDef sym1 e eff', DenoteDef sym2 e eff') => DenoteDef (
5    sym1 + sym2) e eff' where
6    denoteDef :: (DenoteDef sym1 e eff', DenoteDef sym2 e eff') => (+) sym1
7    sym2 e -> Free eff' ()

```

We generalize this denotation function to allow it to operate on definitions from multiple modules, and derive a composition equivalent to the `denote` function (Section 2.4).

```

1  data Program e f = Fragment [e] f
2
3  foldProgram :: (Denote f eff v) => Program (g (Fix f)) (Fix f) ->
4  Program (g (FreeEnv eff v)) (FreeEnv eff v)
5  foldProgram (Fragment defs program) = Fragment (map (fmap foldD) defs)
6  (foldD program)

```

We can now construct a program with definitions and an artificial entry point in the datatype `Program`, which we call a fragment of the program. The function `foldProgram` applies the denotational mapping from syntax to semantics to the whole structure.

```

1  type Envs = FDecl
2  type Eff' v = FunctionEnv (EffV v) (Fix v) + End
3
4  instance DenoteDef FDecl (EnvFree (EffV v) (Fix v)) (Eff' v) where
5    denoteDef = F.denoteDef
6
7  runProgram :: Program (Envs (EnvFree Eff V)) (EnvFree Eff V) -> Out
8  runProgram (Fragment defs exp) = case unwrap
9    $ handle_ defsH (Env { varEnv = [], U.defs = [] } :: Env Eff V )
10   $ denoteDefList defs of
11    (_, env) -> run exp env []

```

We can reuse the effects definitions and the `run` function from the functions component, but we also need `Envs` to define the definition modules in scope, and `Eff' v` to define the effects of the definitions' denotation: one for each populated field of the environment record. The `runProgram` function applies the environment handlers, and *runs* the implementation of the function component entry point.

In this section, we gave the first step of going from processing arbitrary functions component AST fragments, to processing full programs meant as lists of definitions. In the following sections, we will further expand the framework to fit other WebDSL components.

3.2 Global variables

In the previous section, we described how we process function component definitions based on language functions module. Another type of definitions that are relevant for the functions' component is entity definitions. Entities are WebDSL's notion of objects, but they also define the database interface. In WebDSL, global variables are variables that occur at top level of the structure alongside other definitions. They are only allowed to hold values of entity type. In this section, we will introduce entities and the problems they raise, followed by the adaptation of the pipeline presented in Fig. 3.1 to include global variables.

```

1  entity Object {
2    a : Int
3    function foo(b : Int) { return a + b; }
4  }
5
6  var object = Object { a := 1 }
```

Object is an entity definition with a single property `a` describing a database field, and a single function `foo`. In real WebDSL programs, there is much information that can be given in entity definition: constraints on database fields one would expect from a language, validation rules and others.

The variable `object` is an example of global variables. While entities can technically be initialized also within the program AST, they can also be given on the same level as function definitions and in that case, conform to different semantics.

Global variables are initialized before the entry point is called, which puts them closer to program definitions. However, while other definitions are independent of each other, global variables depend on entity definitions being in scope, but also might depend on each other by recursive references.

```

1  entity Recursive {
2    pointer : Recursive
3  }
4
5  var left  = Recursive { pointer := right }
6  var right = Recursive { pointer := left }
```

With the entity `Recursive` we demonstrate this referencing: it has a single property `pointer` which is, entity `Recursive`. The variables `left` and `right` are instantiations of this entity, referencing each other recursively.

When a WebDSL program is run for the first time, the global variables are initialized and written into the database. However, if the database already exists, the variables are only *read* from it, possibly different from the initial definitions, with alteration caused by executing HTTP requests. To represent the program with such semantics, we find it necessary to *process* the global variables in a step separate from denoting both definitions and the entry point.

```

1  data GlobalVar e = VDef VName (EntityDecl e) deriving Functor
2  data VarList e   = VList [GlobalVar e] deriving Functor
3
4  data Program e f g = Fragment [e] (Maybe f) g
```

The datatype `GlobalVar` represents a single variable, and `VarList` a list of all global variables found in the program definition: since global variable depend on each other, they cannot be processed in isolation from each other the way definitions are; the denotation requires multiple *passes* over the list in order to correctly capture the recursive references. We alter the `Program` datatype introduced in Section 3.1.

The program has three parameters: e representing the definitions' type, g representing the entry-point type, and f representing the global variables type. If the g type sum represents a subset of the functions' component, f is equivalent to $\text{VarList} + g$. In `Fragment`, the constructor of `Program`, f is embedded in the `Maybe` monad, to ensure that programs without global variables are clearly marked as such. Alternatively, in case of no global variables, the `VarList` could be carrying an empty list. However, since we embed the `VarList` in the fixed point datatype `Fix` (Section 2.2.2), this solution lacks clarity.

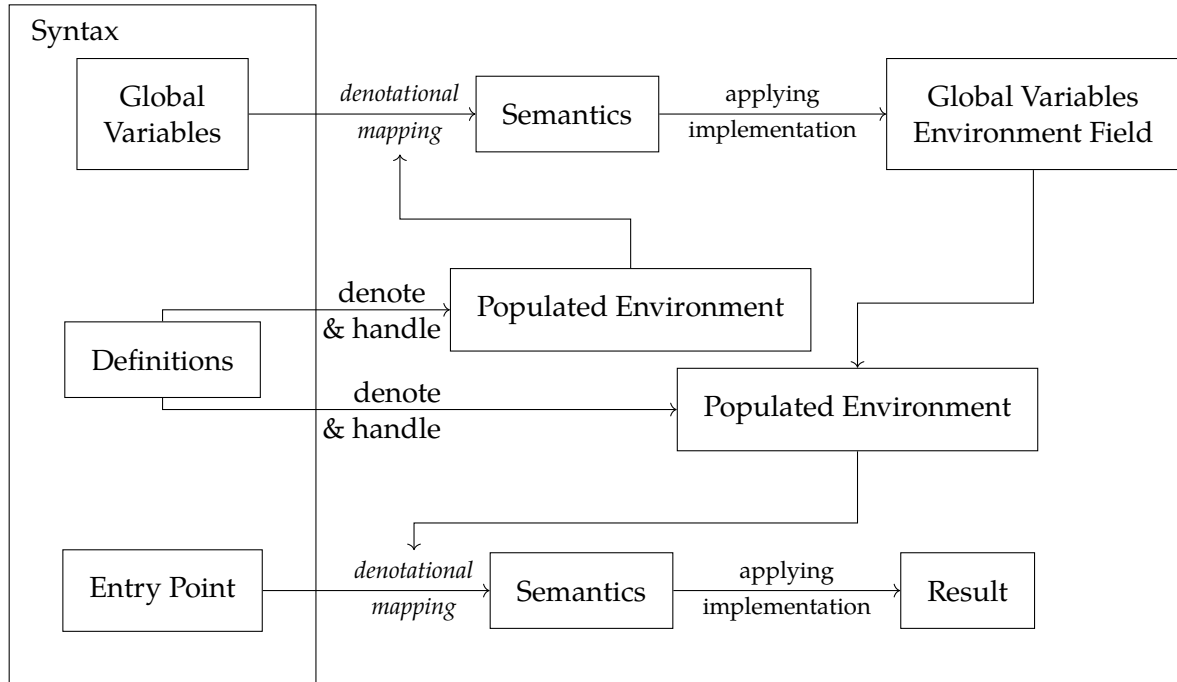


Figure 3.2: Pipeline including Global Variables

In Figure 3.2 we present the full pipeline including pre-processing that we will present in the remainder of this section.

```

1  runProgram :: Program (Envs (Fix Syntax)) (Fix (VarList + Syntax)) (Fix
   Syntax) → Out
2  runProgram (Fragment defs (Just vars) exp) = run (foldD exp) (programEnv
   {globalVars = gVarEnv}) heap
3  where
4    programEnv      = handleDefs (map (fmap foldD) defs)
5    globalEnv       = handleDefs (map (fmap foldD) defs)
6    (gVarEnv, heap) = runVars (foldD vars) globalVarEnv
  
```

The function `runProgram` is an example of executing the denotation of a program with global variables. In comparison with the `run` function from previous Section 3.1, the executable accepts program syntax instead of program semantics. This is necessary, as the global variables component emits different effects than the functions component, so the environment passed to each of the denotation functions is different. The `run` function applies handlers to the function components' effects, while `runVars` applies handlers to the global variables component.

There are some alternatives to this approach. For example, the environment could hold syntax of the definitions instead of their semantics. This would require to explicitly *fold* every retrieved definition into syntax in the denotation mapping, which would pollute the formal

requirements we stated for the mapping (Section 2.4). Another possible solution is taking advantage of the fact that the additional effects occurring in the global variables' are specific to the `VarList` semantics alone, and that component does not *write* any effectful semantics into the environment. On every entry point to the functions' component, the result would have to be `Lifted` inline of the program.

In this section, we explained how we extend the framework and the definition of what a program is to include the global variables' component, which requires a specific processing order in regard to both the program definitions and the program entry point.

3.3 Templates Component

In web programming, templates are used for generating HTML pages and provide reusability that plain HTML code lacks. Within WebDSL, templates component provides high-level abstraction of HTML, semantics for CSS attributes, and at the same time, allow to define the data model of the generated pages. In this section, we explain how the templates component fits in with the functions component, and give the extension to the framework introduced in Chapter 2 that allows to separate the semantics of functions and templates components. In Section 3.3.1 we explain the changes to the syntax component, followed by changes to the denotational mapping in Section 3.3.2. Finally, in Section 3.3.3 we show how this extension fits with the adaptations to the pipeline presented in Section 3.2 and Figure `fig:glo-vars`.

The templates component provides tools for composing HTML websites. It has some effects in common with the functions' component, but it also emits many effects not relevant to the functions component. The entry point to the program is always a template element, which in turn in some strictly defined locations contains entry points to the functions component. We might say that templates component is the *top-level* language of WebDSL and the functions component is *embedded* in it. Because of such strong differences between these two components, we want to retain this division into two sub-languages in the framework.

3.3.1 Syntax

In this section, we will present how we describe the abstract syntax for templates' component. First, we will present some modules from the templates' component, followed by infrastructure analogous to one of the functions' component presented in Section 2.2, and the adjustments to the composition infrastructure.

```
1 data Forms t a = Form t | Label a t | Submit t a deriving Functor  
2 data Layout t a = Header t | String String deriving Functor
```

We present two of the templates component syntax modules. The `forms` datatype describes forms (`Form` constructor), field labels `Label` constructor and buttons (`Submit` constructor). The `Layout` datatype, for the purpose of this example, has two constructors: `Header` for representing HTML `h` tags, and `String` that gives plain text. These datatypes have two parameters, `t` and `a`. We use them to differentiate between the sub-languages: `t` stands for templates component, signifying arbitrary recursion, and `a` represents functions (actions) component, allowing it only to contain syntax modules from the functions' component.

We derive `Functor` type class for the datatype, which allows making `Form t a` a functor over `a`. This is not sufficient - we need to be able to map over *both* of these parameters. The type class that allows us to do that is called *bifunctor*: a functor that allows mapping over two parameters (Awodey, 2006). Bifunctors were hinted to be useful in this style of syntax definition already by Swierstra, 2008, but for different purposes, such as embedding datatypes that already are functors. In our case, the distinction between templates and functions components in the syntax is a result of the necessity of having this distinction in the denotational mapping function and being able to define different effects for these two components.

```

1 instance Bifunctor Forms where
2   bimap :: (a → b) → (c → d) → Forms a c → Forms b d
3   bimap g f (Form body) = Form $ g body
4   bimap g f (Label value contents) = Label (f value) (g contents)
5   bimap g f (Submit action name) = Submit (g action) (f name)

```

Bifunctor is not a derivable type class in Haskell, so we need to provide the implementation ourselves. The function `bimap` is a binary map over bifunctor's parameters, equivalent to **Functor**'s `fmap`.

Bifunctor fixed point datatype

Now, we will attempt to construct a composition of these bifunctor syntax modules.

```

1  formExample :: Fix (Forms (Forms (Layout ()) (Fix Module)) (Fix Module))
   + Layout ())
2  formExample = injF $ Form (Label (S.str "example") (String "text"))

```

With two parameters, it is not possible to conveniently use fixed point `Fix` datatype. In the function `formExample`, we see that `Fix` gives a fixed point only over the last parameter, leaving the first parameter with the same recursion issue that we observed before introducing `Fix` (Section 2.2.2). To remedy this, we introduce `BiFix`: a fixed-point datatype over two arguments.

```

1 data BiFix e f = BIn (e (BiFix e f) (Fix f))

```

`Bifix` has two parameters: `e`, which requires *two* parameters, and `f`, which takes only one argument. In the constructor, `e` is given two arguments, an instance of `Bifix` and an instance of `Fix`. Since the functions component has no notion of the templates component, the second argument should remain being embedded in `Fix`. `BiFix` is a variation of the fixed point `Fix` datatype specific for our use case.

Bifunctor type sum

```

1 infix 6 +:
2 data (f +: g) a b = L' (f a b) | R' (g a b) deriving Functor
3
4 infix 5 <::
5 class f <:: g where
6   inj' :: f a k → g a k

```

We also need a way to compose the two-argument syntax modules, and describe subtyping on them. We will use `+:` bifunctor type sum infix for composition and `<::` infix for bifunctor subtyping. Bifunctor `+:` is given three instances of `<`: for inferring subtypes equivalent to the ones described for functor subtyping `<`: (Section 2.2.2).

```

1 injBf :: (f <:: g) ⇒ f (BiFix g h) (Fix h) → BiFix g h
2 injBf = BIn . inj'

```

With bifunctor sum subtyping, we can define injection into `Bifix`, and redefine `formExample` with a more coherent type definition:

```

1 formExample' :: Bifix (Forms +: Layout) Module
2 formExample' = injBf $ Form (injBf $ Label (S.str "example") (injBf $
   String "text"))

```

Equivalent to the approach in function components, we will use smart constructors to limit the infrastructure constructors when writing tests and programs.

```

1 form  :: (Forms <:: f) ⇒ BiFix f g → BiFix f g
2 label :: (Forms <:: f) ⇒ Fix g → BiFix f g → BiFix f g
3 string :: (Layout <:: f) ⇒ String → BiFix f g
4
5 formExample' = form (label (S.str "example") (string "text"))

```

The three smart constructors, `forms`, `label` and `string`, wrap the equivalent constructor with injection into `BiFix`.

3.3.2 Denotational Mapping

In this section, we will explain how we alter the `denote` function for mapping the templates component and the auxiliary infrastructure. First, we will present a naïve denotation example, followed by the infrastructure it raises. Further, in the two subsections, we will reformulate the templates' denotation function to include templates' component environment, and explain how we unify the function and templates components when necessary.

```

1 denoteForms :: (State Seed <: eff', Random String FormId <: eff', State
  FormId <: eff', StreamHTMLOut <: eff')
2   ⇒ Forms (Env eff v → Free eff' ()) (EnvFree eff v) → Env eff v →
  Free eff' ()
3 denoteForms (Form body) env = do
4   seed  :: Seed ← get -- State Seed
5   formId :: FormId ← encode $ "form_" ++ show seed -- Random String
  FormId
6   put formId -- State FormId
7   renderForm formId -- StreamHTMLOut
8   body env
9   renderTag $ TagClose "form" -- Stream HTMLOut

```

The `denoteForms` function gives one of the cases of mapping `Forms` datatype onto semantics. The effects used can be found in Appendix A; each smart constructor is commented with the effect it constructs. Thanks to dividing the language components into two different parameters, we can differ between the effects they emit. All effects emitted by the templates' component are subtypes of `eff'`, and the functions' component and environment are subtypes of `eff`. We will further discuss the relation between `eff` and `eff'` later in this section. Finally, the denotation `Free` monad parametrized by `eff'`, representing the top-level semantics, returns an empty type `()` instead of a value: all important behavior in the templates' component is emitted as effects instead of an end value.

```

1 class DenoteT sym eff eff' v where
2   denoteT :: sym (Env eff v → Free eff' ()) (EnvFree eff v) → Env eff
  v
3   → Free eff' ()
4
5 instance (DenoteT sym1 eff eff' v, DenoteT sym2 eff eff' v)
6   ⇒ DenoteT (sym1 +: sym2) eff eff' v where
7   denoteT a = case a of
8     (L' f) → denoteT f
9     (R' f) → denoteT f

```

We generalize the denotational mapping to the `DenoteT` type class and an instance to pass the mapping down the bifunctor sum type.

```

1 foldDT :: (Denote f eff v, DenoteT g eff eff' v, Bifunctor g)
2   ⇒ BiFix g f → Env eff v → Free eff' v

```

```
3 foldDT (BIn g) = denoteT $ bimap foldDT foldD g
```

We can now fold the program AST. The `bimap` is called with two functions: `recursive foldDT` to fold the templates' syntax modules, and `foldD` (introduced in Section 2.4.1) to fold functions' modules.

Environment Type

So far, the environment we used held only value fields or fields parametrized by the functions' component effects. With the templates' component, we have two new definitions types: pages, which define the subpages of the given website, and templates - callable units of the templates component that define a *part* of the website.

```
1 page root {
2   header { List }
3   for (i : Int from 0 to 5) { foo(i) }
4 }
5
6 template foo(a : Int) {
7   <p> output(a) </p>
8 }
```

In the code snippet above, we present a page definition and a template definition. The `page root` is the main page of the website, called unless the HTTP request specifies otherwise. In our example, it consists of a header `a` and a loop, calling the template `foo` in each iteration. The template `foo` simply outputs the value in HTML paragraph tags.

```
1 data TemplateDef t a = TDef TName [(PName, Type)] t
```

The template definition `TemplateDef` is what will be present in the environment as `TemplateDef (Env eff' ()) (EnvFree eff v)`. Since it needs to be parametrized by the template components' effects `eff'`, we cannot use the functions' environment `Env eff v` to store it without losing the compatibility with functions component. Instead, we define a new environment record:

```
1 data TEnv eff' eff v = TEnv { actionEnv :: Env eff v,
2   templates :: [TemplateDef (TEnv eff' eff v → Free eff' ()) (EnvFree
3   eff v)]
3 }
```

The record `TEnv` (templates environment) has two fields: `actionEnv` which holds the functions' environment, and `templates`, which is a list of template definitions. When needed, we can add new fields to this record without losing backwards compatibility.

```
1 type TEnvFree eff eff' v = TEnv eff eff' v → Free eff' ()
2
3 class DenoteT sym eff eff' v where
4   denoteT :: sym (TEnvFree eff eff' v) (EnvFree eff v) → TEnvFree eff
   eff' v
```

We can give the final version of the template denotational mapping signature. We define type alias `TEnvFree` for the unexecuted semantics of the templates component, which we will use from now on when discussing templates' environment

Calls to Functions' Component

So far, we have explained how we make the distinction between templates' and functions' components, but not how we *unify* them at runtime. Since the free monads representing each

of the components are parametrized by different effects, they require some way of converting from `Free eff v` to `Free eff' v`.

```

1  run :: Free eff v → Env eff v → v
2
3  denoteForms :: (Stream HTMLOut <: eff') ⇒ (TEnvFree eff eff' v) (
4    EnvFree eff v) → TEnvFree eff eff' v
5  denoteForms (Label name body) tEnv = do      -- Label a t
6    name' ← Pure $ run name $ action env
7    renderTag $ TagOpen "label" []           -- Stream HTMLOut
8    renderPlainText (unbox name') True      -- Stream HTMLOut
9    renderTag $ TagClose "label"           -- Stream HTMLOut
10   body env

```

At first, we might want to simply apply handlers to all functions' component's effects inline, which we do here with the `run` function. Since the outcome is a value and not a free monad, we simply need to use the `Pure` constructor to use the monadic bind `←` notation.

This approach works well on surface, if we assume there are no common elements between `eff` and `eff'`, and that these effects' handlers do not produce output values or require information continuity. Both conditions are broken with the subset of functions' component we consider.

```

1  type Eff v = Cond + Abort v + MLState Address v + End
2  type Eff' v = Stream HTMLOut + MLState Address v + End

```

Let's consider the effect rows `Eff` and `Eff'`, representing possible instantiations of the `eff` and `eff'` variables for some subsets of functions' and templates' component. The `MLState Address v` effect, representing heap of the program, is also emitted by some templates' component denotation. Any correct heap handler also requires information continuity: receiving the current state of heap as input and returning a (possibly modified) heap.

We consider keeping track of the handlers' input and output by the denotational mapping function to be out of scope and polluting the denotation. Because of that, we would like to define a way to handle the effects *partially* and map the remaining effects, relevant to the templates' component, from remainder of `Free eff` to `Free eff'`. This mapping is called *effect masking* (Poulsen and Rest, 2023).

```

1  class (Functor eff, Functor eff') ⇒ Lift eff eff' v where
2    lift :: Free eff v → Free eff' v

```

We define the class `Lift` that gives these expected semantics of. However, for every instance of the framework, a custom `lift` function need to be written, depending on the effects in `eff` and `eff'`. For the purpose of demonstrating our solution, we will be considering the effect rows `Eff` and `Eff'` introduced earlier in this subsection.

```

1  instance Lift (Eff v) (Eff' v) v where
2    lift e = injectSum
3    $ handle funReturn $ handle condition e

```

The two effects of `eff` that need to be handled are `Cond` and `Abort v`, for which we use handlers `condition` and `funReturn`, The outcome is a Free monad of type `Free (MLState Address v + End) v`; we need to extend the type from what it currently is into `Eff' v`. We use the `injectSum` function to stand in for effect masking, but we still need to find an implementation for it.

```

1  type EffReduced = (MLState Address v + End)
2
3  injectSum :: (effReduced <: eff') ⇒ Free effReduced v → Free eff' v
4  injectSum = fold Pure (Op . inj)

```

In first attempt, we build the function with available infrastructure of subtyping syntax `<:`. However, we quickly discover that this implementation cannot be used in our implementation of `Lift`: the inference rules we gave for subtype infix `<:` allow the left-hand side to be a concrete type, and not a functor sum like `EffReduced`.

```
1 infix <<:
2 class (Functor g, Functor h) => g <<: h where
3   cmap :: g k -> h k
```

To allow sum types on the left-hand side of subtyping relation, we define a new infix `<<:` to build a layer on top of the infrastructure we have. We call the injecting function `cmap` (composite map). The type signature of `cmap` looks the same as the one of `inj` (Section 2.2.2), but the inference rules built on this infix will make the difference.

```
1 instance (Functor f, Functor h, f <: h) => f <<: h where
2   cmap = inj
3
4 instance (Functor f, Functor h, f <: h, f' <<: h) => (f + f') <<: h where
5   cmap = \case
6     L f -> inj f
7     R f -> cmap f
```

The first instance is the base case, stating that for every two functors with subtyping relation `<:`, the subtyping relation `<<:` also occurs. The second instance gives the recursive case for left-hand composition: it states that if `f` is a functor that is a subtype of `h`, and `f'` has the `<<:` relation with the supertype functor `h`, then the sum of these two types also has the `<<:` relation with `h`. Both of these cases are given an implementation as a proof of correctness of these semantics.

```
1 injectSum :: (effReduced <<: eff') => Free eff v -> Free eff' v
2 injectSum = fold Pure (Op . cmap)
```

We can now give a correct implementation for the `injectSum` function.

In this section, we explained the infrastructure that allows us to call functions' component elements from templates' component, while retaining structural division between the two classes of modules.

3.3.3 Including definitions and global variables

We explained so far how we alter the framework to support the division of templates' component, but the changes we introduced impact both the definitions and global variables extensions. In this section, we will explain the adjustments necessary for these components. This section uses the same pipeline as presented in Figure 3.2, with main changes including *unifying* the definition types to allow processing templates' and functions' definitions, and assuming a template component entry point.

Definitions

In Section 3.3.2 we gave the datatype of template definitions `TemplateDef`. The template definition, like all other syntax modules of the template's component, is a bifunctor, thus we cannot use the existing definitions' infrastructure to carry it. In this section, we will alter the framework from Section 3.1 to fit the bifunctor modules.

```
1 type TDefs eff eff' v = MLState (TName, [Type]) (TemplateDef (PEnv eff
2   eff' v) (FreeEnv eff v))
```



```

3 denoteDefT :: (TDefs eff eff' v <: f) => TemplateDef (TEnvFree eff eff'
  v) (EnvFree eff v) -> Free f ()
4 denoteDefT template@(TDef name vars e) = do
5   (name :: (TName, [Type])) <- ref template
6   return ()

```

The denotation of template definitions limits to updating the environment. However, the different type signature requires defining a new type class that accepts bifunctors as input.

```

1 class (Functor eff', Bifunctor sym) => DenoteDef' sym f e eff' where
2   denoteDef' :: sym f e -> Free eff' ()

```

The type class `denoteDef'` gives us the type signature we need.

We cannot use the functor `sum +` to compose template definitions with other definitions. However, we still want to include the functions' component definitions in the scope, so we need a way to combine bifunctor- and functor-based datatypes in a single type `sum`.

```

1 data LiftF s t a = LiftF (s a)
2
3 instance DenoteDef sym e eff' => DenoteDef' (LiftF sym) f e eff' where
4   denoteDef' (LiftF x) = denoteDef x

```

To allow the composition, we define the `LiftF` (lift functor) datatype, that allows lifting any functor into a bifunctor. This way, we can infer `denoteDef'` on any lifted functions' component definition and use the bifunctor `sum +:` to compose function definitions.

```

1 type Envs = TemplateDef +: LiftF EntityDef +: LiftF FDecl
2 type Eff'' = TDefs Eff Eff' V + EntityDefsEnv Eff V + End
3
4 instance DenoteDef EntityDef (EnvFree Eff V) Eff'' where
5   denoteDef = E.denoteDef
6
7 instance DenoteDef' TemplateDef (TEnvFree Eff Eff' V) (Free Eff V) (Eff
  '') where
8   denoteDef' = T.denoteDefT
9
10 runProgram :: Program (Envs (TEnvFree Eff Eff' V) (EnvFree Eff V)) () (
  TEnvFree Eff Eff' V) -> Out
11 runProgram (Fragment defs Nothing expr) = case handleDeps defs of
12   (env :: TEnv Eff Eff' V) -> T.run expr env

```

With the given infrastructure, we can define a `Program` with definitions. We can instantiate functions' component definitions with `DenoteDef` type class, and template component definitions with the new `DenoteDef'` class. In the first step of the `runProgram` function, the effects in `Eff''` are handled with the `handleDeps` function. In the second step, it executes the program's entry point, which is a template component; `T.run` is some function that handles the template components' effects in `Eff'`.

Adding Global Variables

The only valid global variables are entity declarations, which are part of the functions' component. Thanks to the separation between entry point, definitions and program entry point, we do not need to alter the `Program` definition further. In comparison with the `runProgram` function from Section 3.2, we need to update the type of definitions and entry point, and use appropriate run functions for the altered types.


```

1  type DefSyntax = Envs (BiFix SyntaxT Syntax) Syntax
2  type SyntaxT = Forms +: Layout +: Page
3
4  runProgram' :: Program DefSyntax (Fix Sym) (BiFix SyntaxT Syntax) → Out
5  runProgram' (Fragment defs Nothing expr) = T.run (bimap foldDT foldD exp
6  ) (programEnv {globalVars = gVarEnv}) heap
7  where
8    programEnv      = handleTDefs (map (bimap foldDT foldD) defs)
9    globalEnv       = handleTDefs (map (bimap foldDT foldD) defs)
10   (gVarEnv, heap) = runVars (foldD vars) globalVarEnv

```

Since the definitions were lifted to be bifunctors, they need a handling function that operates on bifunctors. The global variable component does not need any of the template components' definitions, in fact, the instances of `DenoteT` parametrized by global variables' specific effect row can be left with a minimum (incorrect) implementation, since there is a guarantee that it will never be called:

```

1  instance DenoteT Layout EffG End V where
2    denoteT _ _ = return ()

```

An alternative would be to filter the global variables to assure there are only function component definitions and map them such that they have the correct type: functor sum embedded in `Fix` instead of bifunctor sum of lifted components. This is not very straightforward to achieve: fixed-point datatypes and type sums are designed such that it is easy to aggregate types and inject, but requires a lot of infrastructure to manually remap the types. Both of these solutions have their downsides in unnecessary infrastructure; we decided on the described solution as we find it cleaner and easier to understand.

3.4 Multi-phase Processing

The templates' component provides semantics not only for the rendering of a given website, but also for processing user input data in an integrated manner: an example of that was given in Chapter 1. `WebDSL` implements this by giving two modes of processing HTTP requests, presented in Fig. 3.3.

The first mode, *render*, is utilized when no user input data occurs - it simply generates a static HTML web page by passing through the page's entry point once. The second mode, when input data is provided, adds a data processing step, which consists of three *phases*: *databind* phase, *validate* phase and *action* phase. Each of these phases passes through the page AST once, giving different semantics to the same syntax with each pass. The data processing mode concludes with *redirect*, which renders the new page, or simply *rendering* the same page. In case any validation errors occur, the action phase is skipped, and the current page is rendered with all applicable error messages.

We implement these semantics by defining a different denotational mapping for each of the phases, emitting different effects depending on the semantics of the phase. In the data processing mode, each phase's effect handlers produce some information that needs to be passed as input to the effect handlers of the following phases.

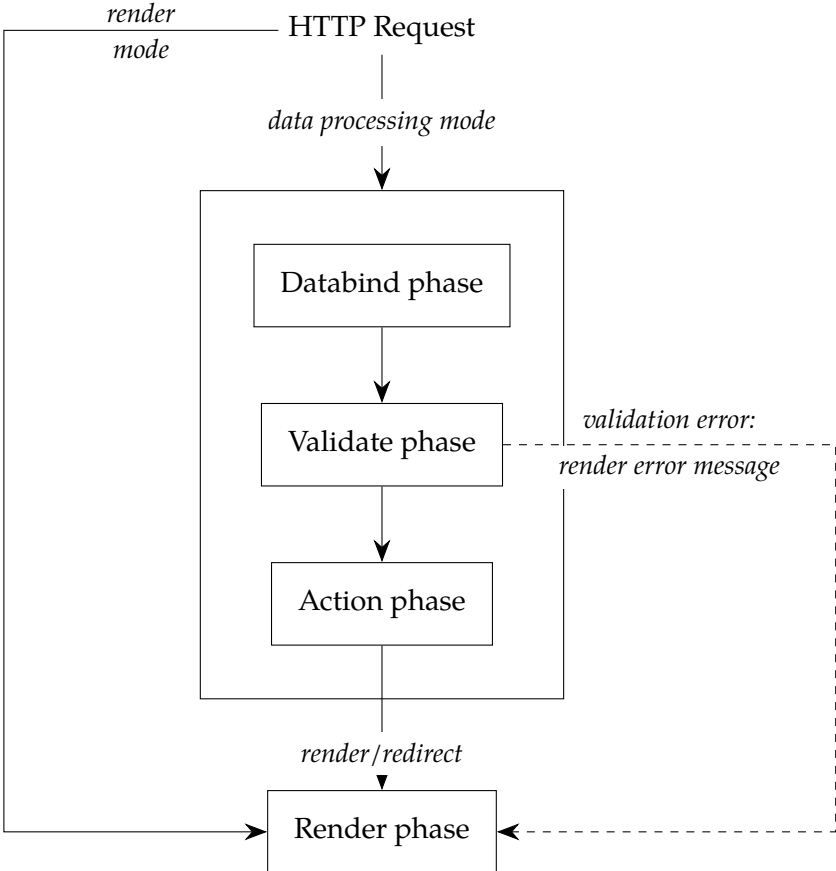


Figure 3.3: Request processing modes

Chapter 4

Evaluation of the Algebraic Effects and Handlers Framework Approach in Haskell

The choice of Algebraic Effects and Handlers framework allowed us to move forward in defining robust denotational semantics for WebDSL. The evaluation of the framework in context of WebDSL semantics can be found in Chapter 5. In this chapter, we will discuss the obstacles we found in using the framework that limit the practical applications of the framework in the form chosen by us, and possible solutions that are out of scope of this project.

In Section 4.1 we will discuss obstacles we found with using and reusing effects and possible ways to fix them, followed by possible improvements to the current phrasing of handling semantics in Section 2.5. Section 4.3 showcases possible improvements to the framework that could be achieved by introducing higher order effects, and Section 4.4 talks about possible benefits or implementing the framework in a language with stronger type system such as Agda. Finally, in Section 4.5 we discuss the limitations of the currently available tools in Haskell when programming within the Algebraic Effects and Handlers Framework.

4.1 Reusable Effects

In algebraic effects and handlers approach, it is desirable to use well-defined, polymorphic effect interfaces, such that the effect parameters are instantiated to specific types when used, and given specific handlers. However, when writing a program with effects, the only way to *identify* an effect being constructed is by its type signature.

```
1 data MLState m v k = Ref v (m → k) | Deref m (v → k) | Assign (m, v) k
```

Consider the effect `MLState`. It has three parameters: `k` representing the continuation, and `m` and `v` that can be specified to more concrete types depending on what the program requires, with `m` representing the *key* of the state and `v` representing a value of the state operations. We will use this effect definition to present the obstacles we found in reusing effects and the possible solutions to these obstacles.

4.1.1 Anonymous Effects

The first issue we want to discuss is the fact that such defined effects are *anonymous*: they are identified by their type signature, and there can exist only one instance of the effect per type. We will first present in detail where this effect occurs, and then discuss two possible solutions and how they compare to each other.

When the `MLState` effect represents the heap of the program, we specify it as given:

```

1  type Address = Int
2  type Heap v' = MLState Address (Fix v')

```

In the Heap type alias, the value parameter v is not fully concrete, but it allows narrowing down the semantics of the effect. The parameter v' represents functor sum of allowed values of the language, not known to a single module's denotational mapping.

The main issue with such definition of effects is that there is no way to specify which effect is used other than by the concrete types of its parameters. Even if we define unique aliases for the effects we use, in Haskell the aliases are only synonyms for the programmer's convenience; both full type and its alias are mutually replaceable.

```

1  type Location      = Int
2  type OtherHeap v' = MLState Location (Fix v')

```

Consider we need a different instance of MLState, possibly because we have different semantics in mind and would like to use different handlers for them; we also want to avoid collisions between the two data structures and be able to use them disjointly - specify only one of them as part of the semantics without having to bring the other effect in scope. Haskell's type system will not recognize Heap and OtherHeap as two different effects, instead will use the two synonyms as stand-ins for *the same* effect instance.

This problem can be solved either on effect definition level or effect construction level.

Definition-level Solution

```

1  data HeapType      = Heap | OtherHeap
2  data MLState' n m v k = Ref' n v (m → k) | Deref' n m (v → k)
3  | Assign' n (m, v) k

```

In the definition-level solution, we redefine the effect datatype to include an additional parameter n . This parameter acts as a *name* of the effect being constructed, with possible values such as `'Heap` or `'OtherHeap`, which can be used as type-level enums with Haskell's DataKinds extensions.

We will try to improve this formulation and discuss possible implications of this design decision. First, we observe that the *name* parameter is not required for any of the handler semantics; we could try to remove it from the effect constructors as redundant information.

```

1  data MLState'' n m v k = Ref'' v (m → k) | Deref'' m (v → k)
2  | Assign'' (m, v) k

```

For this purpose, we present the effect MLState''. Since we use smart constructors to encapsulate operations, it is possible to use this effect, but not without concessions. We will present the possible solutions on the example of smart constructor for the operation Ref.

```

1  ref :: MLState' n m v <: f ⇒ n → v → Free f m
2  ref n val = Op $ inj $ Ref' n val Pure
3
4  ref' :: forall n m f v. MLState'' n m v <: f ⇒ n → v → Free f m
5  ref' _ val = Op $ inj (Ref'' val Pure :: MLState n m v (Free f m))
6
7  refHeap :: forall m v f. MLState'' HeapType m v <: f ⇒ v → Free f m
8  refHeap val = Op $ inj (Ref'' val Pure :: MLState'' HeapType m v (Free f m))

```

The first version of the smart constructor, `ref`, uses the MLState' effect and Ref' taking n as one of the input arguments: the most generic approach. In the second version, `ref'`, we use the Ref'' definition without n argument. Nevertheless, the smart constructor still

need to take an argument of that type as input to specify which effect is being invoked. Even though the effect appears as a constraint of the smart constructor, that is not visible outside the constructor implementation, and using such smart constructor without explicitly passing `n` as value makes it impossible to specify which *name* of the effect do we want to construct. Thus, generic effect requires us to pass the *name* type as value in the denotation. So, even though we can remove `n` from operation constructor, we must nevertheless use it explicitly when invoking the smart constructor.

Finally, we could make non-generic smart constructors such as `refHeap`, which constructs only the effect parametrized by `HeapType`. If we decide for this style, we gain clearer, less cluttered function calls when invoking the smart constructor, but it comes with the downside of loss of generics. For every instance of this effect, regardless if the parameters' collision occurs or not, there needs to be three constructors defined. We could also define a set of semi-generic smart constructors with `n` instantiated to the empty type `()` to use when we do not expect the parameter collision to occur:

```
1 ref'' :: forall m v f. MLState'' () m v <: f => v -> Free f m
2 refHeap val = Op $ inj (Ref'' val Pure :: MLState'' () m v (Free f m))
```

The smart constructor `ref''` is an example of such smart constructor. This solution also is not ideal: in case the previously unforeseen parameter collision occurs in the future, all instances of the smart constructor for the *old* colliding effect instance need to be updated.

After presenting all implication of this approach, it is clear that while solving some problems, requires trade-offs when applied. Additionally, perhaps the *name* parameter should not be a part of an effect signature at all, as it has nothing to do with the effect definition and needlessly pollutes it.

Construction-level Solution

```
1 newtype Location = Loc Int deriving Eq, Num
2 type OtherHeap v' = MLState Location (Fix v')
```

Instead of modifying the whole effect datatype, we can decide to use the *newtype* construct on one of the parameters in place of type aliases. This solution gives us backwards compatibility for the cost of encapsulation associated with using a *newtype*. We must explicitly derive all type classes we need from the type we use, and in case we need to use the *value* of `Location` in the denotation, we need to pattern match on the datatype or explicitly invoke the constructor. These additional operations are irrelevant for the denotational mapping, making the solution imperfect.

Discussion

Having more instances of an effect with the same type signature is a problem we have encountered when working on the framework. Between the two solutions, we use the *construction-level* approach due to less compatibility issues with existing codebase. However, we do find this solution to be imperfect, with *newtype* definitions functioning as a band-aid afterthought to a problem that should have a more systematic solution. A well-designed system with reusable effects that is intended for larger-scale programs should treat this problem as inevitable to occur.

We find that in a system built with effect type collisions in mind, there should be a way to specify the particular effect instance with the type signature alone. This is not possible within the current formulation of the framework in Haskell. Some variant of this is possible with Haskell's `TypeApplications` extension, which we make use of within the smart constructors, but it is not useful for specifying the type of the smart constructor, since it encapsulates the

effect used. The closest systematic approach is defining all effects with a *name* parameter as presented in the definition-level solution, which can be simply ignored if not necessary.

The problem with anonymous effects has been observed before, with solutions requiring framework-level alternations. Lexically scoped handlers allow effects to be single out by name and *lexically* bound to an enclosing handler in languages with parametric polymorphism (Biernacki et al., 2019). This solution should not be confused with lexically scoped handlers implemented in the Effekt language, which operate under contextual polymorphism (J. I. Brachthäuser, Schuster, and Ostermann, 2020). Another solution extending the idea of lexically scoped handlers is algebraic effect instance scopes, which provides the distinction between instance names and instance variables (Balik, 2022). However, it is not clear how possible it is to implement of these calculi within our framework.

4.1.2 Too Many Parameters

In this section, we will discuss how parameters required by the effect constructors are sometimes redundant from the denotational mapping point of view and pollute the denotation with seemingly unnecessary operations.

The `MLState` effect is well-defined: all parameters occur in each of the effect’s operations. We also use the `MLState` effect to abstract interaction with the environment. However, the operations of the `MLState` effect encapsulate the semantics we use imperfectly. Consider the function definitions’ field of the environment record:

```
1 data FDecl e = FDecl FunName [ArgName] e
2 defs :: [FDecl FunName [ArgName] e]
```

Functions are not keyed by an integer value assigned by the handler, but by its name stored in the `FunName` parameter, which is known independently of the handler’s implementation. Therefore, perhaps a different set of operations would be more useful for interfacing the operations with this environment field. We will present an effect that gives the simpler semantics and explain the issues it raises.

```
1 data NamedState m v k = Set v k | Get m (v → k)
```

The proposed effect would have three parameters, the continuation type `k`, `m` representing the key, and the value parameter `v`. The effect has two operations, `Set` and `Get`. The `Set` operation fills in the role of the `Ref` operation, but with one main difference: where `Ref` returned the key value to the continuation, `Set` does no such thing. The name of the function is not relevant for the semantics, so we omit it from the operation. The `Get` operation is equivalent to `MLState`’s `Deref`.

With such definition, we soon discover that a similar problem occurs as we described in Section 4.1.1 when discussing the definition-level approach. Since the parameter `m` does not occur in the constructor, we have to somehow use it anyway to be able to fully define the effect instance used.

```
1 set :: forall m v f. (NamedState m v <: f) ⇒ v → Free f ()
2 set value = 0p $ inj (Set value (Pure ()) :: NamedState m v (Free f ()))
```

Consider the smart constructor `get`. Even though it does technically have a correct type, when calling the function, there is no way to pass information about what type do we expect `m` to be.

```
1 set' :: forall m v f. (NamedState m v <: f) ⇒ Maybe m → v → Free f ()
2 set' _ value = 0p $ inj (Set value (Pure ()) :: NamedState m v (Free f ()))
```

The easiest way to mitigate that is to add another argument of type `Maybe m` to the smart constructor, allowing to pass information about type without an instance of that type by

using the **Nothing** constructor. However, this action places us where we started: we need to manage this irrelevant argument in the denotation function.

```

1  denote function = do
2    _ :: FunName ← ref function      -- specification by output type
3    set' (Nothing :: FunName) function -- specification by input type

```

In the code snippet above, we see the comparison of using these two approaches for defining smart constructors, both forced to carry unnecessary information. The first style, specification by output type, is more readable and clear in providing semantics. Here we see that even if we have a *single* instance of an effect in scope, it might be challenging to construct.

We can still consider a solution from the previous section: instead of using the most generic smart constructors, we could specialize them based on key types. We expect environment definitions to be keyed by either their names *or* their names and argument types, so this specialization requires less smart constructors than one per effect instance, making this approach more feasible.

```

1  setS :: forall m v f. (NamedState String v <: f) => v → Free f ()
2  setS value = Op $ inj (Set value (Pure ())) :: NamedState String v (Free
3    f ())

```

The smart constructor `setS` is an example of a specialized smart constructor. This approach still leaves the information about type in the denotation function, this time in the smart constructor name.

The best solution would occur if it was possible to somehow specify which effect is being used by using a *type hint* describing the effect invoked. This is not possible to achieve in Haskell and would probably require language support for effect semantics or a more specialized type system.

4.1.3 Conclusion

In this section, we presented multiple limitations of algebraic effects as defined in our framework for providing clear and readable denotational mapping. None of these limitations make it impossible to use the framework, but they do introduce *quirks* that one must get used to in order to work with the framework. If these limitations were solved systematically on the framework level, the programs written in the framework would require less boilerplate code and be simpler. However, it is not clear how possible it is to solve these limitations, other than swapping one imperfect solution for another.

4.2 Composite Handlers

In this section, we will discuss how effects with partially overlapping operations could be treated to avoid code duplication.

When discussing an effect describing state, there are three different variants to consider: a read-only state (Reader effect), a write-only state (Writer effect) and a read-write state which comprises the operations of the other two variants.

```

1  data Reader v k = Read (v → k)
2  data Writer v k = Write v k
3  data State v k = Get (v → k) | Set v k

```

In the code snippet above we present the implementation of these three effects. Even though the `State` effect is the composite effect, it is useful to limit the available operations to *only read* or *only write* to better capture the semantics under modelling, as the same *side effect* might be restricted differently depending on, for example, the phase of execution (Section

3.4). However, this means that the implementation of the side effect has to be defined three times, as each effect datatype requires its own handler.

```

1  read  :: Foldable t => t v -> (v, t b)
2  write :: Foldable t => t v -> t v
3
4  readH = Handler_ { hdlr_ = \(Read k), state -> k v state'
5    where (v, state') = read state }
6  writeH = Handler_ { hdlr_ = \(Write v k), state -> k state'
7    where state' = write state }
8  stateH = Handler_ { hdlr_ = \op, state -> case op of
9    (Get k) -> k v state' where (v, state') = read state
10   (Set k) -> k state'   where state'     = write state }

```

This leads to a unification issue: the only way to even partially mitigate that and unify the implementation across effect datatypes, is to define a *reading* function (`read` in the code snippet above) and a *writing* function (`write` in the code snippet above) and call them in appropriate cases of the handlers' implementation. In the code snippet, the `hdlr_` fields of the three handlers give examples of the expected style. This approach is not secure enough: it might lead to these three handlers being decoupled from each other in the future, if one of the handlers' implementation is changed instead of changing the unifying reading *and* writing functions at the same time.

A more secure framework-level solution would allow to specify a single handler for multiple effects, keeping the implementation together even if the operations are separated. This solution is implemented in some algebraic effect systems, for example in the Scala Effekt library (J. Brachthäuser, 2019). We shall refer to these multi-effect handlers as *composite* handlers. With our current framework, it is not possible to define this kind of handlers: instead of functor sum, it would require a functor **or** operation: a handler handling *any subset* of the effects it is parametrized by.

While it is possible to define some version of composite handlers in the current Haskell framework, not all the semantics described would be easy or possible to define.

Having a single handler for the separate reading and writing effects would also allow discontinuing the use of the `State` effect datatype, since its semantics are fully captured by the sum of the other two datatypes. Alternatively, with composite handlers, it would be particularly useful to be able to define an effect as a sum of other effects. These semantics could be fulfilled by the functor sum syntax:

```

1  type State v = Read v + Write v

```

The type alias `State` shows an example of such multi-effect approach. Its validity depends on the implementation of the composite handlers, which we do not have at the moment. An important thing to take into account is that the effect row `State v + End` is not the same thing as `Read v + Write v + End`, as the functor sum syntax is right associative. The former is equivalent to `(Read v + Write v) + End`, while the latter is equivalent to `(Read v + (Write v + End))`. Further, due to the right-associativity, using the introduced `State` type alias could impact the `<`: constraint, making the `Read` and `Write` effects not inferrable separately from each other.

4.3 Applying Higher-order Effects in the Framework

Higher-order effects are a level of abstraction over algebraic effects, allowing to model *non-algebraic* effects. These effects are distinct in that they require handlers to be applied as part of their semantics. Higher-order effects provide a construct called *elaborations* which are a

pre-processing step transforming higher-order effect into a combination of algebraic effects, and handlers applied inline (Poulsen and Rest, 2023). In this section, we will present a use case of higher order effects discuss how higher order effects could improve the framework.

Within the denotational mapping, we do have a case where applying the handler inline is necessary: the interactions with environment. We explain the issue in detail in Section 2.4.2. To summarize, the `MLState` effect for interacting with environment would have to be parametrized by the effect row type itself, which would require recursive type definitions. Since the solution is to *apply handlers inline*, using Higher-Order effects looks like a promising alternative solution. For the framework of this section, we use the Haskell definitions from Diepraam, 2023 (Chapter 3), which are based on the (Agda) framework presented by Poulsen and Rest, 2023

```

1  derefEnv :: (Functor eff) => VName -> Env eff v -> Free eff Address
2  derefEnv'' name env = do
3    (loc, env) <- handle_ environment env (deref name)
4    return loc

```

The function `derefEnv` provides an encapsulation for applying handler inline to dereference a variable name. Other operations on environment include *assigning* name and value pairs to the environment, *referencing* definitions, *populating* the environment in case of function and entity function calls, and finally *clearing* parts of environment when entering a new scope.

```

1  data EnvAction v e f k
2    = Get String e (v -> k)           -- equivalent to Deref
3    | Update (String, v) e k         -- equivalent to Assign
4    | Put v e k                       -- equivalent to Ref
5    | UpdateAll [(String, v)] e k    -- multiple consecutive Updates
6    | Clear e k

```

The datatype `EnvAction` provides a hypothetical Higher-Order effect interface as described in the previous paragraph. The parameter `v` Represents the value being put in the environment, `e` represents the type of environment (it could be either `Env` or `TEnv`). The parameter `f`, representing the computation type, is not used; higher-order effects are typically used to operate on *computations* instead of avoiding type recursion as we do here. Nevertheless, this parameter needs to be present as it specifies the algebraic effects that the `Hefty` monad elaborates into. Finally, `k` represents the continuation of the computation.

```

1  eEnvAction :: EnvAction v (Env f v) (Free f) (Free f Address)
2  eEnvAction (Get varName env k) = do
3    (loc, env) <- handle_ environment env (deref name)
4    return loc

```

In the function `eEnvAction` we provide a case for elaborating the `Get` operation of the higher-order effect `EnvAction`. The implementation of the function is generally identical to the `derefEnv` function, the difference lies in it's signature. We can now attempt to rephrase the framework to include this elaboration.

```

1  type Eff = MLState Address (Fix V) + End
2  type Eff' = EnvAction (Fix V) (Env Eff V) (Free Eff) :+ Lift (MLState
3    Address (Fix V))
4  type Syntax = Eval
5  run :: Hefty Eff' V -> V
6  run program = unwrap $ handle_ heap $ elab (eEnvAction ^ eLift) program

```

In the run function above, we present a general run function that operates on a program embedded in the `Hefty` monad and operating on Higher-Order effects. The algebraic effects are described by `Eff` and the Higher-Order effects are described by `Eff'`. Notice how the algebraic effect in scope must also be *lifted* in `Eff'`.

This section gave a flavor of introducing Higher-Order effect in the framework. The main benefits of this approach is having a clearer encapsulation of environment operations. Currently, they are standalone functions, and embedding the handling operations of environment in higher-order effects solves the problem of recursive type definitions. The `EnvAction` effect differs from other stateful effects in that the effect operations take the environment as argument instead of encapsulating it. This difference stems from the use case (described in Section 2.4.2): passing the environment as value in the denotational mapping gives a static scope by construction, and alternative approach would require explicit removal of values from the environment.

A practical improvement to the referenced higher-order effects framework would require lifting semantics for algebraic effects that would define and elaborate a lift on a whole effect row (e.g. `LiftSum (Eff)`) over manually lifting and elaborating the effects one-by-one (`Lift (MLState Address (Fix V))`).

The application of Higher-Order effects in this denotational semantics, at current state of completion, is limited to environment operations. As such, it was considered too low priority for the scope of the project.

4.4 Stronger Type System of a Dependently Typed Language

Large part of the design of the framework we use is focused on operations on types to allow the expressiveness necessary. Even though the framework *works* in Haskell, in many places it forces awkward phrasing due to nearing to the limits of Haskell's expressiveness. In many cases, the type inference system of Haskell fails, requiring explicit type signatures to functions. Additionally, *within* the function implementations, additional *type hints* are necessary to guide the interpretation of function calls and effect interpretation. Many type system extensions have been adopted to make this framework pass Haskell's static verification.

Using a language with stronger typing will make defining the framework undoubtedly more difficult, as more things that Haskell makes possible to work, have to be defined more explicitly in languages with stronger type systems. Nevertheless, Higher Order and Algebraic effects have been encoded in dependently typed languages such as Agda (Poulsen and Rest, 2023). The Data Types a la Carte findings that we use as a basis of our framework also has been translated to Agda (Rest et al., 2022). These encodings require more boilerplate code due to Agda's requirements of guaranteed termination and strictly positive types. Things that are given *for free* in Haskell, require extensive background in Agda. For example, the strictly positive implementation of functors in Agda is done via containers (Abbott, Altenkirch, and Ghani, 2005).

Despite all this trouble, there are some clear benefits of implementing such framework in Agda. Having the framework in a language with native support for theorem proving would allow to prove some properties of the framework to ensure its correctness. It would also allow to prove some properties about the algebras to demonstrate the completeness of the denotation.

The dependent types feature of Agda will not be particularly useful for the framework in its current shape. The core element of the framework, functor sums, is essentially an opposition to dependent types: instead of strengthening the guarantees on the type, functor sums loosen that relation.

4.5 Tooling

The final obstacle we discuss is lack of tooling for programming with the chosen paradigms. As mentioned in the previous section, many of them go beyond what Haskell is made to do to and typically used to do. This is reflected in tooling available for working in Haskell.

HLint is a linter for Haskell: tool for suggesting possible improvements and simplifications to the code (Mitchell, 2017). The linter provides useful suggestions for *typical* functional programming improvements, but it is not capable of suggesting constraint optimizations on functions. If a certain effect is considered to be used in a module’s denotational mapping, and later discontinued, it remains to the attention of the programmer to remove the constraint of that effect being part of the effect row. Obsolete constraints can propagate up to the framework definition and only systematically reviewing the type signatures allows to keep the effect row to the minimum. This issue is particularly impactful if some parts of the denotation need to be updated in the future, increasing the effort required to keep the denotation up to date with the compiler.

Another tool that is not sufficient for supporting the framework is the GHC type errors: when concerning subtyping constraints or the functor sum, the messages are often imprecise, incorrect, or describe the *lack* of a certain component instead of specifying what is missing.

Missing Effect Subtype Constraint

Whenever an effect smart constructor is used within denotational mapping, it raises a subtype constraint on the denotation function’s effect row. Consider the denotation of the *if/then/else* ternary operator with the smart constructor of the `Cond` effect:

```
1 denote :: (Functor eff, Lit Bool <: v) => Boolean (EnvFree eff (Fix v))
   → EnvFree eff (Fix v)
2 denote (If c a b) env = do
3   c' ← c env
4   cond (projV c') (a env) (b env)
```

If the `Cond <: eff` constraint is not part of the constraint tuple, the following error is rised:

```
1 Overlapping instances for Cond <: eff arising from a use of ''cond
```

The error later refers to the inference instances defined in subsection 2.2.2. Instead of specifying the underlying cause of the error, which is lack of the constraint being specified for the function, the error message focuses on an artifact of the framework definition.

Missing Component in the Functor Sum

Whether it is a type sum specifying the syntax modules in scope (e.g. `Boolean`, `Arith`), values (such as `Lit Bool`, `Lit Int`), or the effect row, the denotation function or the program might invoke a subtype constraint that is not actually fulfilled. Consider the following program:

```
1 program :: Fix (Arith + Expr)
2 program = if' true (int 1) (int 2)
```

The smart constructor `if'` gives the ternary *if/than/else* operation which is part of the `Boolean` syntax module. When the module is missing from the type sum in the signature of the program, the following error is given:

```
1 Couldn't match type ''Expr with 'g0 + g''0 arising from a use of 'if''
```

The *source* of the error is lack of the `Boolean` component in the functor sum, but the error message only specifies the *last* element in the relevant type sum, `Expr`. In case the type sum is defined further away from the location of the error, it is even more difficult to make the connection of which piece of code is missing. The same type of issue with the error message occurs if an effect is missing in the effect row type alias, or the effect has incorrect parameters in comparison with the constraint raised by denotational mapping.

Overall, the tooling support of the current shape of the framework is not perfect. While it is possible to successfully work with the framework, especially for an experienced Haskell programmer, the learning curve related to reading error messages and optimizing the framework is a significant factor. This is a concern especially for the expectation of the denotation being easy to update when changes to WebDSL semantics occur.

Chapter 5

Case study of WebDSL: Evaluation

Defining the denotational semantics of WebDSL gave us insight necessary to evaluate algebraic effects and handlers as an effective tool for that purpose. In this chapter, we will describe the outcome of this evaluation. First, we will compare the framework with earlier approaches at modelling WebDSL in Section 5.1. Second, we will evaluate the capabilities of algebraic effects and handlers in providing sufficient modularity to model different aspects of WebDSL in Section 5.1, followed by the discussion the possible improvements to WebDSL's semantics based on the framework in Section 5.3. Finally, we will try to give an answer to the research questions presented in Chapter 1 in Section .

5.1 Comparison with Other Approaches to Model WebDSL's Dynamic Semantics

Before this denotational model of WebDSL, two other attempts where made. They successfully covered a subset of denotational semantics but ultimately the projects where abandoned due to increased difficulty of covering a larger language subset. The first of these attempts was a different approach in Haskell, based on the usage of the `State` monad. The second attempt was undertaken in Redex, a DSL for modelling semantics. In this section, we will compare the three approaches to evaluate our chosen approach.

It is not very easy to compare these three approaches one-to-one, as each of them covers a slightly different subset of WebDSL, allows different semantics within the model itself, and takes different liberties with the WebDSL compiler. Therefore, we will compare different qualitative and quantitative aspects of the model: lines of code, model coverage of the WebDSL semantics, and finally modularity and extendability of the models.

5.1.1 Lines of Code

LOC is an imperfect metric of comparison. It does not prove anything about the completeness of the semantics, but instead tells us about the efficiency of the model (how much infrastructure is required to model this particular language in the given framework) as well as how many auxiliary definitions are necessary. For example, the Haskell State framework comes with a parser into the syntax representation used for the model that takes approximately 550 lines. Our model achieves the same functionality with syntax smart constructors, taking circa 250 lines.

Another way LOC are polluted is that our approach combines framework *and* denotation elements, while both previous approaches work on an existing framework - the (very lightweight) definition of State monad is imported from Haskell's library, and Redex as a DSL does not need additional framework infrastructure. Where Haskell's state approach requires just the State monad, multiple effects and instances of them are used in our approach.

Table 5.1: Approximate LOC of the denotational models

	Haskell (State Monad)	Redex	Haskell (AE & Handlers)
syntax definition	125	100	170
parsing semantics	550	-	250
denotation	700	300	1400
tests	1000	250	2700
total	2500	750	6900

However, this increased amount of definitions is the necessary cost of more expressive and clear semantics, so it simply a matter of offset between LOC and other measures of the framework quality.

In Table 5.1 we present the approximate LOC for the different models. The values do not add up exactly, as each of the three models had elements that are more difficult to categorize or not shared between the models. It is important to note that in our approach, we considered readability of the code more important than optimization of LOC value. While functional programming style allows excessive line lengths, the average line length in our project is much lower than in the Sate Monad model. It is one of the reasons for such a big difference in size of the project. The Haskell State Monad approach includes test files written in plain text compatible with the parser. Redex does not include any parser.

The algebraic effects and handlers approach lists syntax smart constructors as parsing semantics, which are not included in the syntax count. The denotation count for includes the denotational mapping of each module and the effect definitions and smart constructors. The test includes tests, framework definitions they use and handler definitions.

It can be clear on first sight that Redex allows for most concise tests based on the language definitions, while both Haskell approaches require more infrastructure, particularly for the algebraic effect and handlers approach. However, comparing tests based on LOC metrics is imperfect in comparing what amount of either test coverage of the semantics definition or test coverage of full WebDSL semantics. We included handlers as part of the test count, however in the state monad some elements of the denotation might be considered implementation due to lack of clear separation between them. Even when considering boilerplate, Algebraic Effects and Handlers model has 74 tests, while the State Monad 45. The Redex model has 60 tests. We are not able to meaningfully compare the coverage of the tests in comparison with WebDSL semantics.

The clear conclusion that we can make based on this metric is that the Algebraic Effects and Handlers approach is the biggest of the three models. The LOC metric alone is not enough to make draw any more conclusions; they would be only guesses. We will move on to other metrics to put this conclusion into a more useful perspective.

5.1.2 Coverage

The second metric we will consider is the coverage of WebDSL semantics. In Table 5.2 we present the comparison between different models. We use the same component classification as within our model, and only considering the elements that at least one of the models included in their semantics. Among the three models, the Redex model is the smallest, which corresponds to the LOC metric for the model. We find that perhaps the DSL is well suited to model a general-purpose language like the functions component, but it is not flexible enough to extend it to model the templates component.

Between the two Haskell approaches, the State Monad model seems to have a slightly bigger coverage in templates component and includes more components overall. The asterisks in the column signify that some semantics associated with that module are incomplete or incorrect, pointing at incorrectness or lack of expression capabilities of the modelling ap-

proach. For example, the **return** semantics (associated with functions module) do not correctly model aborting execution of the AST. The entities are limited to properties, so none of the entity functions' semantics are modelled.

Table 5.2: Semantics coverage between the models

	Haskell (State Monad)	Redex	Haskell (AE & Handlers)
functions component			
expressions	✓	✓	✓
lists	✓	✓	✓
variables	✓	✓	✓
functions	✓*	✓	✓
loops	✓	✓	✓
loop filters			✓
entities	✓*	✓	✓
database interaction	✓	✓	✓
global variables			✓
templates component			
attributes	✓*		✓
template calls	✓		✓
page calls	✓		✓
layout & render	✓		✓
multi-phase processing	✓		✓
forms			✓
validation semantics	✓		✓
action semantics	✓*		✓
sessions	✓		
ajax actions	✓		

It is also important to point out that the Haskell State monad often operates on *desugared* syntax, missing some expressions occurring in WebDSL. For example, large part of the phases semantics is tied to forms, but the State Monad model only considers explicit functions related to each of the data processing mode phases. The State monad approach also misses some elements of the functions component, prioritizing the broadness of the model over its depth. We draw the conclusion that our model is the most detailed out of the three models. Based on the test counts from the previous sections, we also can say that Redex is the best tested out of the three models, and perhaps the framework that makes writing tests the easiest out of the three.

5.1.3 Modularity and Extendability

Modularity gives the possibility to *zoom in* on a part of the language and inspect it in separation from other components of the language. It also impacts extendability: how easy it is to extend a particular module of the language or add a new component to the model? There are different framework aspects in which modularity and extendability can be considered: framework, dependency of the semantics on each other, and finally the modularity of the multi-phase semantics.

First of these aspects is the environment, by which we understand the scope-dependent state.

The Redex framework does not use a notion of environment. Instead, it relies on a global substitution function which operates on all *language extensions*. If a new extension is added that requires substitution, the global function needs to be updated. However, other aspects

of the environment (e.g. function definitions) can be done by extending *reduction relations*, allowing to separate different concepts within the environment.

In the Haskell State Monad approach, the different aspects of the environment are passed through the various evaluation functions as separate variables. This impacts the extendability - every new *type* of information added to the environment will require updating all functions' instances, which is non-negligible.

In the Algebraic Effects and Handlers approach, extending the environment requires minimal changes. Adding a field to the record is fully backwards compatible. Because all interactions with a certain field go via the same handler, altering the implementation of the environment is also trivial.

The second aspect of modularity is the independent state of the semantics. In the Haskell State Monad, all state information is contained in a single tuple, so any state operation requires handling other, irrelevant information. In Algebraic Effects and Handlers framework, this is one of the elements that is clearly solved with more modularity in mind - every aspect of the state is considered separately, as part of a different effect instance. Additionally, it is possible to differentiate between *bi-directional* state and one-directional reader or writer semantics. In terms of extendability, the algebraic effects approach allows full isolation between the effect with no cost of adding new effect instances to the semantics. The State Monad approach requires updating all operations on state if it is to be altered.

When it comes to syntax, while Redex generally allows to divide the language into different *extensions*, only both Haskell approaches provide a clear distinction between the functions and templates components as modules with completely different functions and semantics. The State Monad approach divides them by separating them into two different datatypes. Extending each of the datatypes is possible, and updating the evaluation function of that syntax constructor is fairly easy, but ultimately does not provide the same level of isolation as our framework does.

The multi-phase semantics are another test of modularity. In the State Monad approach, the difference in semantics is given by providing four phase-specific functions as arguments which are called when appropriate. This allows to easily separate the *common* semantics and phase-specific semantics, however at the cost of requiring all phase functions to have the same signature. This means that no further specification of the phases can be inferred from the type signatures. In the algebraic effects approach, the differences in the phases are specified primarily by the algebraic effects in scope, which give a clear description of available operations and differences between the phases. Since the component is missing from the Redex model, it cannot be discussed here.

Overall, while the State Monad approach is much more lightweight and covers a slightly greater set of the language, its design decisions are tailored to the *current* version of the framework, sacrificing future extendability of the model and the lack of division between implementation and semantics might make it difficult to update existing semantics when necessary. The Haskell Algebraic Effects and Handlers framework requires extensive infrastructure to make the model work as expected, but allows a much more coherent, uniform structure that is easy to extend and update in the future.

Both Haskell approaches have the benefits of leveraging the power of general-purpose language when giving semantics of less-standard language features that are abundant in WebDSL, such as the different methods used to model the multi-phase semantics or the bifunctor extension to the language we designed to model the communication between the two components of WebDSL (Section 3.3). In contrast, Redex framework does not have such capabilities, but its clear benefit is the integrated tests that are much easier to write and the support for randomized testing (Klein et al., 2012). Haskell's property-based testing library

QuickCheck (Claessen, 2024) can be similarly utilized for both of the Haskell approaches, but again, more infrastructure needs to be used to generate correct tests.

5.2 (Lack of) Modularity in the Denotation

In this section, we will discuss the cases where the modularity of our framework might be insufficient or raise some concerns. First, we will describe a case where two modules interfere with each other's denotational mapping in the templates component. Second, we will describe the problems raised by adding new value types to the functions' component. Finally, we will reflect on the general phrasing of the model where we found the modularity to be insufficient.

5.2.1 Interaction between modules

In some cases, we tried to decouple two syntax modules from each other to separate their semantics, but even though we managed to encapsulate some semantics in a separate module, they still need to leave some artifact in the original module to ensure correctness. We will use attributes semantics to illustrate this example.

In web templating, attributes are the way to relate to CSS elements within HTML. Web-DSL adds its own semantics on top of that:

```

1  template foo(a : String) {
2    section[attributes ["foo"]]{ output(a) }
3  }
4
5  template bar { foo[foo="foo1"]("bar") }
```

In the code snippet above we give an example template definition with template call. The template `foo` has a `section` constructor with additional parameters in square brackets before the actual contents of the section. The square brackets signify operations on attributes; in this case, the `foo` attribute and its value is retrieved from the attribute scope to be attached to the section's HTML tag. To populate the attribute scope, when template `foo` is called in template `bar` definition, the attribute `foo` is defined with its value. The same syntax inside the square brackets has different meaning, depending on the type of constructor it is used.

There is more to attribute semantics than in this example; effectively, they create their own DSL with reach semantics. We would like to encapsulate these semantics to its own module, but we cannot fully separate them from other templates' component's modules that give them more specific meaning.

```

1  data Attributes t a = SelectionList [AttributeSel t a] t
2  type AttName = String
3  data AttributeSel t a = Attributes [a] | AttDef AttName a | ..
```

To describe the attribute's component syntax, we define the `AttributeSel` (attribute selection) and `Attributes` datatypes. `AttributeSel` gives the semantics of each expression that can occur in square brackets, while `Attributes` collects them into a list of expressions. The last parameter of the `SelectionList` constructor is the template element that *contains* these attribute expressions. Note that there are more syntax constructors of the `AttributeSel` datatype, not relevant to this example. The `Attributes` constructor defines fetching attribute-value pairs from the attribute scope as used in the `foo` template, while `AttDef` allows defining a new attribute id-value pair, as in the `bar` template definition.

```

1  denoteSel :: (Lift eff eff' v, State [(AttName, String)] <: eff', Render
   v <: eff') => AttributeSel (PEnv eff eff' v) (FreeEnv eff v)
2  -> PEnv eff eff' v
```

```

3  denoteSel (Attribute e) env = do
4    e'          ← lift $ e actionEnv env
5    (e'' :: String) ← derefH (unbox e') attsH env
6    put [(attName, e'')]
7
8  denoteSel (AttDef attName e) env = do
9    e' ← lift $ e $ actionEnv env
10   e'' ← render e'
11   put [(attName, e'')]

```

The attributes' scope is one of the environment fields, accessed by the attsH handler, and the denotation of `AttributesSel` puts attribute id-value pairs to the State `[(AttName, String)]` effect. This is not enough to fully realize the attribute's component semantics, as they depend on the syntax that surrounds them.

```

1  bar :: Bifix (TemplateDef :+ Attributes :+ TemplateCall) Str
2  bar = tDef "bar" (injBf $ Attributes [AttDef "foo" (str "foo1")])
3    (tCall "foo" [(str "bar", String)]) )

```

The variable `bar` holds the template `Bar` definition from our example. We can decouple the attributes' module from others with syntax, but it still leaves some artifacts to handle in denotation:

```

1  denote :: forall eff eff' v v'. ( MLState Address v <: eff, Lift eff eff'
   ' v, MLState Address v <: eff', State [(AttName, String)] <: eff') =>
   Page (PEnv eff eff' v) (FreeEnv eff v) → PEnv eff eff' v
2  denote (TCall name args Nothing) env = do
3    (TDef tName params body) ← derefH (name, map snd args) templatesH env
4    env' ← populateEnv lift (actionEnv env) (map fst
   params) (map fst args)
5    attributes ← get
6    body env { actionEnv = env'', attributes = attributes }

```

In this simplified version of the template call denotation, we need to include some elements of the attributes' semantics, as visible on line 5 of the code snippet. Even if we do not want to include the `Attributes` module in the language subset we are considering, the semantics necessary for proper definition of attributes' component.

This is the crux of the issue with our framework we are trying to showcase here: even if we can achieve structural modularity by separating syntax into different modules, and defining the denotation of the syntax in separate functions, there is still some interference between different denotational functions that does not have a clear source. Looking at this denotation of the template call, we cannot know for sure what gives rise to this particular phrasing of attributes, only by browsing the whole model we can find where the attributes' semantics are defined. Another aspect of this interconnection is that when the `Attributes` module was retroactively added to the model, it required updating the denotation of already defined denotation functions across many modules. We find it likely that when refining the semantics in the future, more of such cases will rise, when one module will impact other existing modules, and some errors might be introduced to the model if not all affected modules are updated according to the requirements of the newly added module.

It might be tempting to consider attributes another sub-language as the functions component is, by using a trifunctor to define the syntax. We consider this unnecessary, as its semantics are not very rich: they limited to a single environment field and very limited effects. The attributes would still have to be embedded in On the other hand, the overhead in syntax definitions would be too big to justify for a component that is encapsulated in a single syntax module with low chance of ever being extended in the future.

5.2.2 Introducing new literals

Extending the functions component might rise similar problems in the denotation function. Different types and multi-type binary operations are separated into different modules, for example, the `Arith` module for integers and `Boolean` module for boolean values hold operations related to one type, but the `Expr` module holds multi-type binary operations. If we want to extend the model by some other types that are present in the language (Email, date types, or even `Double's`), it will require not only specifying the new module, but also updating existing modules.

This issue with semantics being *shipped off* to the `Expr` module could have been avoided if each module would define its own notion of all the relevant operations, however that would introduce much more repetition in the denotation, with the same operation (e.g. comparison) having different constructors for each of the modules that implement the operation.

5.2.3 Top-level solution

Another example where modularity failed was when global variables were introduced in the model, which we describe in Section 3.2. This is not a case of an inter-module issue; global variables required redefining the top-level function of the framework by requiring an additional processing step, which only added complexity in later stages.

5.3 Possible Improvements to WebDSL

In this section we will discuss the possible improvements to WebDSL semantics that we found by constructing the model.

In the functions component, the database can be both read and written in any phase. We model this by providing two different effects: the `ReadDb` and `WriteDb` effects. The reading is required by lazy loading of entities: any entity referred to by the program can be present in the database of the program, but not defined by the program itself. As long as any function is executed, the reading effect must be present in scope.

The database writing is not necessarily an operation that should be available at any stage to the programmer. The only two cases where the database needs to be populated is the initial processing of global variables, if there are not in scope yet (Section 3.2), and the `action` phase of the data processing mode (Section 3.4), if the data being written is properly validates. Allowing database writes in other cases introduces an insecurity within the user input validation semantics. Therefore, it would be useful to separate these actions and disallow database writes in the rendering phase.

5.4 Evaluation of the model

In this section, we will evaluate the model with the research subquestions stated as the basis of this project. The questions can be found in Section 1.1.

5.4.1 How much of WebDSL can we cover within the model?

In the artifact to this thesis, we give the model with a subset of WebDSL semantics, covering most of the functions' and templates' components, and the data model, detailed in Table 5.2. This limited coverage is the result of limited scope of this project, and not some intrinsic limitation of the framework.

Based on D. Groenewegen, 2023 (p.29, Table 2.1), those are the three main components of the language, with the rest of the components built on top of these three. We present the table from D. Groenewegen, 2023 in Table 5.3, extended by coverage column, describing which of

Table 5.3: WebDSL language concepts coverage

Concept	Functionality	Coverage
data model	data entity objects with database persistence primitive, reference, and collection types load/save functionality for objects	✓
ui templates	pages connected by navigation links render HTML tags and data model values forms with databind update data model objects	✓
functions	general-purpose object oriented language actions triggered from ui templates update data model objects with assignments	✓
queries	query data model objects in functions	
email	email templates, based on ui templates send email trigger in functions	
data validation	validation phase after databind in ui templates data model invariants, functions assertions render messages in ui template	✓
access control	rule-based sublanguage to create security policy declare principal data model object rule checks can use expressions from functions rule needs to refer to existing ui templates	
native classes	declare interface of Java code in data model create objects and invoke methods in functions	
services	ui templates page request to generate JSON read incoming JSON request data in functions	
search	search field mapping in data model search queries in functions	
JS CSS embed	embed JS and CSS fragments in ui templates	
AJAX updates	update subset of ui templates inside page	

these components are covered within the model. Since the three base components of the language are covered, we conjecture that *almost all* the WebDSL semantics can be covered in the model. The *native classes* component raises our doubts, as modelling pure Java might be out of reasonable scope for denotational semantics.

5.4.2 To what extent can we employ the effects paradigm to make the model modular, such that the components are independent of each other?

In Section 5.2, we discussed the limits of the framework regarding modularity. In some cases, interference between modules cannot be avoided, as given by the example of the Attribute module in Section 5.2.1 and this interference might grow with the count of syntax modules included in the language. However, in comparison with other modelling attempts discussed in 5.1, it is clear that the modularity provided by Algebraic Effects and Handlers interface contributes greatly to the existing modularity, allowing the model independence of components found in the Redex framework, with the flexibility and expression of State Monad framework, giving a model that is easy to extend in its current state. Despite that, we cannot exclude that adding some components mentioned in 5.2 will require going back to previously defined effects and possibly denotation to improve the existing tools. For example, the queries component might require more expressive database reading semantics.

5.4.3 How does the algebraic effects and handlers framework compare to the previous attempts of modelling WebDSL's dynamic semantics, in terms of maintainability and conciseness?

In Section 5.1, we compared the framework to previous attempts of modelling WebDSL's dynamic semantics. While the denotational mapping considered in separation of the framework provides conciseness of expression, the whole framework requires much more boilerplate to work and be used effectively, compared with the other frameworks. This boilerplate includes smart constructors required by both effect and syntax datatypes, as well as very detailed type signatures of denotational mapping components and handlers that are necessary for these components to be compiled correctly. In terms of maintainability, we find the framework performing better than the other frameworks, evidenced by more opportunities for easier extandability provided by better modularity of the framework.

5.4.4 Does these semantics give rise to any possible improvements of WebDSL?

Yes; in Section 5.3, we describe the changes we found when analyzing the model that could improve the soundness of the program. These improvements are related to database access in different parts of the model. We postulate limiting writing to the database to the initial pre-processing of global variables and the data processing mode, which allow sufficient security of altering the database.

Chapter 6

Related work

This section discusses similar and related work to this presented in the thesis.

Algebraic Effects

Algebraic effects were first defined in the field of category theory by (Plotkin and Power, 2001), to be later adapted into multiple functional programming languages as libraries (Xie and Leijen, 2020) or first-class language features (J. I. Brachthäuser, Schuster, and Ostermann, 2020). Since algebraic effects model only a subset of all possible algebraic effects, they were later extended to Algebraic Effects and Handlers (Plotkin and Pretnar, 2009), which allowed to model non-algebraic components as handlers. We make use of that extension as the basis of our framework. Further developments to the framework includes scoped effects (Wu, Schrijvers, and Hinze, 2014), which also allow to express some higher-order effects. A more general approach than scoped effects is *higher-order* effects (Poulsen and Rest, 2023). We consider these higher-order effects as a possible improvement to our framework in Section 4.3.

Free monad

The encoding of effects as a Free monad was taken from (Swierstra, 2008). The paper presents a robust way of achieving modularity in Haskell, which can possibly extend to other languages with sufficiently rich type systems, and provides the encoding of algebraic effects as a free monad. Specialised variants of the Free Monad used for representing higher order effects includes Latent Effects' Tree (Berg, Schrijvers, et al., 2021) and the Hefty Monad (Poulsen and Rest, 2023; Berg and Schrijvers, 2024). We make use of for the possible higher-order effects formulation in Section 4.3.

Formal Semantics Frameworks

There exist multiple tools that allow defining formal semantics of programming languages, often built for vastly different purposes. One of such frameworks is the K Framework (Roşu and Şerbănută, 2010), which is focused on modeling details of concurrency, and as such has been used to give a complete semantics model of Java (Bogdanas and Roşu, 2015). Since concurrency is not a relevant part to WebDSL semantics, we found the framework focus not suited for our purpose.

Another notable framework is Redex (Klein et al., 2012), which was found to be particularly useful in supporting the verification the correctness of given calculi. A previous attempt at modelling WebDSL semantics was conducted in Redex and found to be unsuccessful and the framework insufficient for the task, which we discuss in Section 5.1. Because of that, we also do not consider the framework as a possible tool for our task.

Interaction trees

An alternative approach to executable denotational semantics is presented by Xia et al., 2022, which uses interaction trees also based on the free monad, but defined coinductively. The interaction trees' framework primary goal is to reason about effectful computations, but main applications discussed have much lower-level focus of web operations (Koh et al., 2019).

Chapter 7

Conclusion

We have presented an algebraic effects and handlers framework that we use to give denotational semantics of WebDSL. The denotation is not complete. This was never a goal due to the size and complexity of the language. However, we gave nearly complete semantics to a few core components: the functions' component, the templates' component, and the multi-phase processing of the templates' component.

Additionally, we give evaluation of the chosen approach to algebraic effects and handlers and describe issues encountered and suggestions for undertaking similar projects with our chosen approach.

Finally, we compared our framework to other approaches of modelling the denotational semantics of WebDSL and give recommendations for refining the semantics of the language's compiler.

7.1 Future Work

There are a few areas in which improvements or extensions can be made.

The first and most obvious aspect is expanding the denotational model by incorporating more of WebDSL's components. For the model to serve its purpose as a documentation tool for the language users, the current support of the language is not sufficient.

The second area which gives promising benefits is using the model to generate oracles to test WebDSL's compiler with the use of Haskell's property-based testing library QuickCheck.

To support both the framework and the property-based testing, a compiler from WebDSL to the AST representation of the syntax could be added, together with the annotations produced by WebDSL's static verification. Additionally, a parser from unambiguous syntax AST into ready-to-compile WebDSL projects would allow to fully automate the property-based testing of the language.

Finally, Haskell's tooling in the shape of linter software and compiler's error messages could be improved to provide more support for working with heavy constraint-based code-base and type sums.

Bibliography

- Abbott, Michael, Thorsten Altenkirch, and Neil Ghani (Sept. 2005). “Containers: Constructing strictly positive types”. In: *Theoretical Computer Science* 342.1, pp. 3–27. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2005.06.002. URL: <http://dx.doi.org/10.1016/j.tcs.2005.06.002>.
- Awodey, Steve (2006). *Category theory*. Oxford University Press.
- Bach Poulsen, Casper (June 2023). *Algebraic Effects and Handlers in Haskell*. URL: <http://casperbp.net/posts/2023-07-algebraic-effects/index.html>.
- Balik, Patrycja (2022). “Algebraic Effect Instance Scopes”. MA thesis. University of Wrocław.
- Benton, Nick, John Hughes, and Eugenio Moggi (2002). “Monads and Effects”. In: *Applied Semantics*. Springer Berlin Heidelberg, pp. 42–122. ISBN: 9783540456995. DOI: 10.1007/3-540-45699-6_2. URL: http://dx.doi.org/10.1007/3-540-45699-6_2.
- Berg, Birthe van den and Tom Schrijvers (May 2024). “A framework for higher-order effects & handlers”. In: *Science of Computer Programming* 234, p. 103086. ISSN: 0167-6423. DOI: 10.1016/j.scico.2024.103086. URL: <http://dx.doi.org/10.1016/j.scico.2024.103086>.
- Berg, Birthe van den, Tom Schrijvers, et al. (2021). “Latent Effects for Reusable Language Components”. In: *Programming Languages and Systems*. Springer International Publishing, pp. 182–201. ISBN: 9783030890513. DOI: 10.1007/978-3-030-89051-3_11. URL: http://dx.doi.org/10.1007/978-3-030-89051-3_11.
- Biernacki, Dariusz et al. (Dec. 2019). “Binders by day, labels by night: effect instances via lexically scoped handlers”. In: *Proceedings of the ACM on Programming Languages* 4.POPL, pp. 1–29. ISSN: 2475-1421. DOI: 10.1145/3371116. URL: <http://dx.doi.org/10.1145/3371116>.
- Bogdanas, Denis and Grigore Roşu (Jan. 2015). “K-Java: A Complete Semantics of Java”. In: *ACM SIGPLAN Notices* 50.1, pp. 445–456. ISSN: 1558-1160. DOI: 10.1145/2775051.2676982.
- Brachthäuser, Jonathan (2019). *Scala effekt: Multiple handlers at once*. URL: <https://b-studios.de/scala-effekt/guides/multiple-handlers.html>.
- Brachthäuser, Jonathan Immanuel, Philipp Schuster, and Klaus Ostermann (Nov. 2020). “Effects as capabilities: effect handlers and lightweight effect polymorphism”. In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA, pp. 1–30. ISSN: 2475-1421. DOI: 10.1145/3428194. URL: <http://dx.doi.org/10.1145/3428194>.
- Claessen, Koen (2024). *QuickCheck: Automatic testing of Haskell programs*. URL: <https://hackage.haskell.org/package/QuickCheck>.
- Diepraam, Terts (2023). “Elaine: Elaborations of Higher-Order Effects as First-Class Language Feature”. MA thesis. Delft University of Technology.
- Felleisen, Matthias, Robert Bruce Findler, and Matthew Flatt (July 2009). *Semantics Engineering with PLT Redex*. en. The MIT Press. London, England: MIT Press.

- Groenewegen, D.M. (Nov. 2023). “WebDSL: Linguistic Abstractions for Web Programming”. PhD thesis. Delft University of Technology. doi: 10.4233/uuid:fb0cc4b7-a67b-474b-9570-96eb054a39ec.
- Groenewegen, Danny M., Elmer van Chastelet, Max M. de Krieger, and Daniel A. A. Pelsmaecker (2023). “Eating Your Own Dog Food: WebDSL Case Studies to Improve Academic Workflows”. In: *Eelco Visser Commemorative Symposium (EVCS 2023)*. Ed. by Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann. Vol. 109. Open Access Series in Informatics (OASICS). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 13:1–13:11. ISBN: 978-3-95977-267-9. doi: 10.4230/OASICS.EVCS.2023.13. URL: <https://drops.dagstuhl.de/entities/document/10.4230/OASICS.EVCS.2023.13>.
- Groenewegen, Danny M., Elmer van Chastelet, Max M. de Krieger, Daniel A. A. Pelsmaecker, and Craig Anslow (2023). “Conf Researchr: A Domain-Specific Content Management System for Managing Large Conference Websites”. In: *Eelco Visser Commemorative Symposium (EVCS 2023)*. Ed. by Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann. Vol. 109. Open Access Series in Informatics (OASICS). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 12:1–12:6. ISBN: 978-3-95977-267-9. doi: 10.4230/OASICS.EVCS.2023.12. URL: <https://drops.dagstuhl.de/entities/document/10.4230/OASICS.EVCS.2023.12>.
- Klein, Casey et al. (Jan. 2012). “Run your research: on the effectiveness of lightweight mechanization”. In: *ACM SIGPLAN Notices* 47.1, pp. 285–296. ISSN: 1558-1160. doi: 10.1145/2103621.2103691.
- Koh, Nicolas et al. (Jan. 2019). “From C to interaction trees: specifying, verifying, and testing a networked server”. In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs. CPP ’19*. ACM. doi: 10.1145/3293880.3294106. URL: <http://dx.doi.org/10.1145/3293880.3294106>.
- Leijen, Daan (2023). *The Koka programming language*. URL: <https://koka-lang.github.io/koka/doc/book.html>.
- Lindley, Sam, Conor McBride, and Craig McLaughlin (2017). *Do be do be do*. arXiv: 1611.09259 [cs.PL]. URL: <https://arxiv.org/abs/1611.09259>.
- Mitchell, Neil (2017). *HLint: Haskell source code suggestions*. URL: <https://github.com/ndmitchell/hlint>.
- Mosses, Peter D. (1990). “Denotational Semantics”. In: *Formal Models and Semantics*. Elsevier, pp. 575–631. doi: 10.1016/b978-0-444-88074-1.50016-0. URL: <http://dx.doi.org/10.1016/B978-0-444-88074-1.50016-0>.
- Plotkin, Gordon and John Power (2001). “Adequacy for Algebraic Effects”. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, pp. 1–24. doi: 10.1007/3-540-45315-6_1.
- (2002). “Notions of Computation Determine Monads”. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, pp. 342–356. doi: 10.1007/3-540-45931-6_24.
- Plotkin, Gordon and Matija Pretnar (2009). “Handlers of Algebraic Effects”. In: *Programming Languages and Systems*. Springer Berlin Heidelberg, pp. 80–94. doi: 10.1007/978-3-642-00590-9_7.
- Podnieks, Karlis (2018). “Philosophy of Modeling: Neglected Pages of History”. In: *Baltic Journal of Modern Computing* 6.3. ISSN: 2255-8950. doi: 10.22364/bjmc.2018.6.3.05. URL: <http://dx.doi.org/10.22364/bjmc.2018.6.3.05>.
- Poulsen, Casper Bach and Cas van der Rest (Jan. 2023). “Hefty Algebras: Modular Elaboration of Higher-Order Algebraic Effects”. In: *Proceedings of the ACM on Programming Languages* 7.POPL, pp. 1801–1831. doi: 10.1145/3571255. URL: <https://doi.org/10.1145/3571255>.
- Rest, Cas van der et al. (Oct. 2022). “Intrinsically-typed definitional interpreters à la carte”. In: *Proc. ACM Program. Lang.* 6.OOPSLA2. doi: 10.1145/3563355. URL: <https://doi.org/10.1145/3563355>.

- Roşu, Grigore and Traian Florin Şerbănută (Aug. 2010). “An overview of the K semantic framework”. In: *The Journal of Logic and Algebraic Programming* 79.6, pp. 397–434. ISSN: 1567-8326. DOI: 10.1016/j.jlap.2010.03.012.
- Stachowiak, Herbert (Dec. 1974). *Allgemeine Modelltheorie*. de. New York, NY: Springer.
- Staton, Sam (2010). “Completeness for Algebraic Theories of Local State”. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, pp. 48–63. ISBN: 9783642120329. DOI: 10.1007/978-3-642-12032-9_5. URL: http://dx.doi.org/10.1007/978-3-642-12032-9_5.
- Swierstra, Wouter (Mar. 2008). “Data types à la carte”. In: *Journal of Functional Programming* 18.04. DOI: 10.1017/s0956796808006758.
- Szamotołski, Marcin (July 2018). *From free algebras to free monads*. URL: <https://coot.me/posts/free-monads.html>.
- Wu, Nicolas, Tom Schrijvers, and Ralf Hinze (Sept. 2014). “Effect handlers in scope”. In: *SIGPLAN Not.* 49.12, pp. 1–12. ISSN: 0362-1340. DOI: 10.1145/2775050.2633358. URL: <https://doi.org/10.1145/2775050.2633358>.
- Xia, Li-yao et al. (2022). “Executable Denotational Semantics with Interaction Trees”. AAI29257237. PhD thesis. USA. ISBN: 9798351434490.
- Xie, Ningning and Daan Leijen (July 2020). *The Effect library and benchmarks for the paper Effect Handlers in Haskell, Evidently*. DOI: 10.1145/3410241.

Acronyms

AST abstract syntax tree

IR intermediate representation

DSL domain-specific language

LOC lines of code

Appendix A

Effects used in the model

In this chapter, we will present the effects we use in our denotation.

A.1 Functions' component's effects

In this section, we will list the effects first introduced in the functions' component.

A.1.1 Cond

```
1 data Cond k = Cond Bool k k
```

`Cond` is an effect we use to model choosing a computation branch. It has one parameter, the continuation `k`, and one constructor `Cond` with three arguments: `Bool` holding the branching condition and two branches.

A.1.2 Abort

```
1 data Abort v k = Abort v
```

`Abort` is an effect for aborting the current computation. It has two parameters: `v` representing the value to be returned instead of the remaining computation, and `k`, the computation type. The effect has a single constructor `Abort` with one argument representing the value.

A.1.3 MLState

```
1 data MLState m v k = Ref v (m → k) | Deref m (v → k) | Assign (m, v) k
```

`MLState` is an ML-style state definition, where the state holds a map of key-value pairs. The effect has three parameters: `m`, the type of the key, `v`, the type of the value, and `k`, the type of the continuation. It has three constructors. `Ref` gives the operation of adding a value type to the environment, which returns some key to the computation. `Deref` gives the opposite operation, where based on a key, a value is returned to the computation. `Assign` constructor allows to overwrite a value stored at a given key: it accepts a key-value pair, and nothing is passed to the computation.

A.1.4 Random

```
1 data Random e v k = Random e (v → k)
```

Random is an effect for generating some random value based on a seed input. It has three parameters: *e*, the type of seed value, *v*, the type of random value, and *k*, the computation type. It has a single constructor **Random**, which takes the seed value and returns the random value to the computation.

A.1.5 DBWrite

```
1 type Uuid = String
2
3 data DbWrite v k = SetEntity (EntityDecl v) k | SetVar (VName, Uuid) k
```

DBWrite is an effect for writing to the database. It has two parameters, *v* giving the type of values, and *k* giving the continuation type. It has three constructors. **SetEntity** saves an entity to the environment. **SetVar** saves a global variable as a pair of variable name and an unique `Uuid` that defines an entity.

A.1.6 DBRead

```
1 typ Uuid      = String
2 type IsSuccess = Boolean
3 type VName    = String
4
5 data DbRead e k = GetEntity Uuid (e → k) | GetAll EName ([e] → k) |
  Connect (IsSuccess → k) | LoadVariables [(VName, Uuid)] → k
```

DBRead is an effect for reading from the persistent data structure. It has two parameters, *e*, the type of entity values stored, and *k*, the continuation type. It has four constructors. **GetEntity** allows to retrieve an entity by its unique `Uuid` value. **GetAll** allows to retrieve all entities of a certain type. **Connect** allows to establish a connection with the database. It returns a boolean value to the continuation, informing of the success of the database connection. **LoadVariables** allows to retrieve all global variables.

A.1.7 MutateEnv

```
1 data MutateEnv env k = DropLocalVars env (env → k) | LiftObjectEnv env
  env (env → k)
```

MutateEnv is a denotation-only effect (handled inline) for operations on effects. It has two parameters, *env* – the environment type, and *k*, the continuation type. Because environment is parameterized by the effect row, the effect cannot be part of it. This effect has two constructors. **DropLocalVars** is used for dropping variables that become inaccessible on scope change. It takes an environment as argument and returns it to the computation. **LiftObjectEnv** is used for lifting functions properties to regular variables when calling entity functions. It has three arguments: the *global* and *local* (entity) environment, and the computation which accepts an argument as input.

A.2 Templates' component's effects

In this section, we will list the effects raised by the templates component.

A.2.1 Stream

```
1 data Stream t k = Out t String k
```

`Stream` is an effect for writing to multiple output streams. It has two parameters, `t` representing the enum value defining streams of the effect, and `k` – the continuation type. The effect has a single constructor `Out` with three arguments: the enum type `t`, `String` with the value being written, and the continuation `k`.

A.2.2 State

```
1 data State v k = Get (v → k) | Put v k
```

`State` is a single-cell state formulation with two parameters, `v` the value being stored, and `k` the continuation type. It has two constructors: `Get`, which outputs a value to the computation, and `Put`, with two arguments, the value to store `v` and the continuation.

A.2.3 Render

```
1 data Render v k = Render v (String → k)
```

`Render` is an effect representing rendering values to some string descriptions. It has two parameters: `v`, the value to render, and `k`, the continuation type. The effect has a single constructor `Render`, which takes an argument value `v`, and returns a `String` type to the continuation.

A.2.4 Reader

```
1 data Reader m v k = Read m (v → k) | ReadNext (v → k)
```

`Reader` is an effect for read-only state formulation. It has three arguments analogous to `MLState` (A.1.3): `m`, the key type, `v`, the value type, and `k`, the continuation. The effect has two constructors. First of them, `Read`, takes the key as input and returns a value to the continuation. The second constructor, `ReadNext`, does not take any arguments and returns the value argument to the continuation.

A.2.5 Writer

```
1 data Writer e k = Write e k
```

`Writer` is an effect for write-only state formulation. It has two parameters, `e`, the type of the expression to be written, and `k`, the continuation type. It has a single constructor `Write`, which takes an argument of type `e`.

Appendix B

WebDSL syntax modules

In this chapter, we will describe the syntax modules used for providing WebDSL syntax.

B.1 Functions' component syntax

In this section, we will describe the syntax components of the functions' component. The components of this syntax are functors, with a single parameter representing the type of children nodes in the AST.

B.1.1 Arith

```
1 data OpArith = Add | Div | Sub | Mul | Mod
2
3 data Arith e = LitAr Int | OpArith OpArith e e
```

`Arith` is a module giving arithmetic literal and arithmetic operations. It has two constructors, `LitAr`, giving the literal `Int`, and `OpArith`, giving binary arithmetic operations. `OpArith` takes three arguments, the enum `OpArith` defining available arithmetic operations, and two children representing the two values in the operation.

B.1.2 Bool

```
1 data OpB = Or | And
2
3 data Boolean e = LitB Bool | OpB OpB e e | If e e e
```

`Bool` is a module giving boolean literal and boolean operations. It has three constructors. `LitB` gives the boolean literal `Bool`. `OpB` gives boolean binary operations, analogous to arithmetic `OpArith`, parameterized by the enum `OpB`. `If` gives the ternary *if/then/else* operation with three children nodes.

B.1.3 Expr

```
1 data Cmp = Eq | Neq | Lt | Lte | Gt | Gte
2
3 data Expr e = OpCmp Cmp e e
```

`Expr` is a module giving comparison operations. It has a single constructor, `OpCmp`, parameterized by the enum `Cmp`.

B.1.4 Str

```
1 data Str e = LitS String | Add e e | Length e
```

Str is a module giving string and operations on strings. It has three constructors. `LitS` gives the string literal. `Add` is a binary operation that gives string addition. `Length` is an unary operation that gives the length of string values.

B.1.5 Col

```
1 data Col e = LitC [e] | OpIn e e | LComp (Maybe OpB) e VName e [Filter e ]
```

Col is a module giving collections and operations on collections. It has three constructors. `LitC` gives the collection recursive type of lists. `OpIn` gives the *contains* binary operation.

`LComp` gives the list comprehension. It has five arguments. The boolean enum `Maybe OpB` represents a possible fold on the comprehension output. The second argument represents operations on a single list element. The third argument of type `VName` represents the name to which the list element is bound in the second argument. Fourth argument gives the list which is the basis of the list comprehension. Finally, a list of filters defines some possible transformations of the list value before the comprehension is applied.

```
1 [ (x + 1) | id : Int in (1..10) offset 3 ]
```

In the code snippet above, we present an example of WebDSL list comprehension with a single filter applied (`offset`).

B.1.6 Filter

```
1 type IsAscending = Bool
2
3 data Filter e = Where e | OrdBy e IsAscending | Limit e | Offset e
```

Filter gives some simple operations on lists. It has four constructors: `Where`, which gives Haskell **filter** operation, `OrdBy`, which allows to order the list elements both ascending and descending, depending on the value of the `IsAscending` argument `Limit` gives the Haskell **take** operation, and `Offset` gives the Haskell **drop** operation.

B.1.7 Eval

```
1 data Eval e
2   = Var      VName      -- use a variable
3   | VDecl   VName      e -- declare but not initialise, with
  continuation
4   | VValDecl VName e    e -- declare and initialise variable
5   | VAssign VName e
```

`Eval` is a module giving variable assignment, declaratrion, and use syntax. It has four constructors. `Var` defines using a defined variable. `VDecl` defines declaring a variable. `VAssign` defines assigning a value to a declared variable. `ValDecl` defines the combination of the previous to operations: declaring a variable and immediately assigning a value.

B.1.8 Fun

```
1 data Fun e = Return e | FCall FunName [e]
```

Fun is a module giving operations related to functions. It has two constructors. Return gives function return with a value in the child node. FCall defines a function call of a function with name FunName and function arguments in the second argument's list.

B.1.9 Stmt

```
1 data Stmt e = S e e
```

Stmt gives statement concatenation. It has a single constructor S, with two arguments: a statement and its continuation.

B.1.10 Loop

```
1 data Loop e = ForCol VName e e [Filter e] | ForArith VName e e e | While
  e e
```

Loop is a module giving different loops. It has three constructors. ForCol gives a collection iteration loop. ForArith gives an integer range-based loop. While gives a while loop with two arguments, the stopping condition and the body.

B.1.11 Entity

```
1 entity Object {
2   a : Int
3   function foo(b : Int) { return a + b; }
4 }
5
6 var object = Object { a := 1 }
```

We present a small example of entity definitions and uses. In the WebDSL code snippet above, we give an entity definition (Object) with a single property a and a function foo. We also give an example of instantiating entity of this type in the variable object.

```
1 type PName = String
2
3 data EntityDecl e = EDecl EName [(PName, e)]
4
5 data Entity e = PropAccess e PName | ECall e FunName [e] | PropAssign e
  PName e | PVar PName | Save e
```

Entity is a module giving various operations on objects. PropAccess defines access to properties – variables bound to entities. ECall gives calls to entity functions. PropAssign allows assigning values to properties. PVar is a way to access entity properties from the entity functions, where the object to which the property is bound does not have to be specified. Save allows to save an entity to the database.

EntityDecl is a module defining entity declaration. It has two arguments: the name of the entity type, and a list of arguments bound to names.

B.1.12 VTuple

```
1 data VTuple e = Validate e String [String]
```

VTuple is a module giving validation tuples. It has a single constructor `Validate` with three arguments: the validating expression, the error message to print in case of a validation failure, and a list of object properties mentioned in the validating expression, in case the `VTuple` is attached to an object. The list of object properties is output of static verification, it is not present in validation tuples' syntax of WebDSL.

B.1.13 Redirect

```
1 data Redirect e = Redirect String [(e, Type)]
```

Redirect is a module describing redirection to a different subpage. It has two arguments, the name of the page to redirect, and the list of arguments and their types.

B.1.14 Global variables

```
1 data GlobalVar e = VDef VName (EntityDecl e)
2 data VarList a = VList [GlobalVar a]
```

Global variables only occur at the top level of the language definitions, but they structurally conform to the same principles as functions components. `GlobalVar` datatype defines a single variable with the constructor `VDef`. It has two arguments: The variable name and the entity declaration. `VarList` gives a full list of global variables. It has a single constructor `VarList`, giving a list of `GlobalVar` datatype.

B.2 Templates component

In this section, we will list the syntax modules of the templates' component. These datatypes are bifunctors, parametrized by `t` giving the child nodes that are other bifunctor template components, and `a`, which are entry points to the functions' component.

B.2.1 Layout

```
1 type ClassName = String
2 type IsAttAssigned = Bool
3
4 data Layout t a
5   = Header IsAttAssigned t
6   | Title String
7   | Section IsAttAssigned t
8   | String String
9   | Block IsAttAssigned (Maybe ClassName) t
```

Layout gives some basic HTML layout elements. It has five constructors. The `IsAttAssigned` parameter in some of the constructors keeps track of if the syntax element is directly enclosed in an attribute module.

`Header` defines elements enclosed in the header tags `h1`, automatically keeping track of the header size based on nesting in components of type `Section`. `Title` gives page titles which are part of HTML `head` definition. `Section` defines elements enclosed in `span` tags. `String` defines plain string elements. `Block` encloses its continuation in `block` tags.

B.2.2 Render

```

1 data Xml t a = Xml String (Maybe (a, Xml t a))
2
3 data Render t a = XmlR (Xml t a) | Output a | Raw a

```

Render is a module giving layout elements with entry points to the functions' component. It has three constructors. `XmlR` gives a plain text XML where attribute values can be entry points to functions component. `Output` and `Raw a` entry point functions with the only difference that the former escapes HTML character, and the latter does not.

B.2.3 Stmt

```

1 data StmtT t a = S t t

```

`StmtT` is a datatype giving statement concatenation, analogous to the functions' component definition (B.1.9).

B.2.4 Loop

```

1 data LoopT t a
2   = ForCol VName a t [Filter a] -- for all elements in collection
3   | ForArith VName a a t -- for numbers in range from e2 to e3
4   | While a t

```

`LoopT` is a module analogous to the function's component formulation of loops (B.1.10). The only difference that it provides a distinction between functions' entry points and *bodies* of the loops, which are templates.

B.2.5 Attributes

```

1 data Attributes t a = SelectionList [AttributeSel t a] t
2
3 data AttributeSel t a = Attribute AttName | AllAttributes [AttName] |
  AttDef AttName a

```

`Attributes` is a datatype with a giving a list of operations selecting attributes for the template element that is stored in the second argument of its constructor. `AttributeSel` datatype gives the attribute selection operations. It has three constructors. `Attribute` selects attributes from scope based on the name. `AllAttributes` selects all attributes from scope except ones defined in it's argument list. `AttDef` defines a new argument independent of the attributes' scope.

B.2.6 Page

```

1 data Page t a = PNavigate PgName [a] String | TCall TName [(a, Type)] (
  Maybe t) | Elements

```

`Page` module gives elements related to page navigation. It has three constructors. `PNavigate` defines links that are rendered in the page's layout. `TCall` defines *template function* calls, with three arguments: the name of the template, the arguments to that template definition, and an expression that is evaluated at retrieval in the scope of the template location. `Elements` constructor is the retrieval flag for the template call's third argument.

B.2.7 Forms

```
1 data Forms t a = Form IsAttAssigned' t | Label a t | Submit t a
2
3 data Input r t a = Input r Type
```

Form and Input datatypes are at the core of the multi-phase semantics discussed in Section 3.4. Forms has three constructors: Form enclosing its continuation in form tags, Label defining a label tag before its continuation, and Submit, defining a button with an action to perform in the first argument and a name in the second argument.

B.2.8 Phases

```
1 data Databind t a = Databind a
2 data Validate t a = Validate a String
3 data Action t a = Action a
```

In this section, we present three datatypes that define actions that occur *only* in the phases they are named after.

B.3 Definitions

In this section, we will present the syntax modules used for defining various elements that are later called. Some of these variables are functors and belong to functions' environment, others are bifunctors and belong to the templates' environment. They can be distinguished by the amount of arguments.

B.3.1 Functions

```
1 data FDecl e = FDecl FunName [ArgName] e
```

FDecl gives the function declaration that is invoked by FCall (B.1.8).

B.3.2 Entities

```
1 type EName = String --entity name
2 type PName = String --property name
3 type Props = [(PName, Type)]
4 data ImplicitProp = Id
5 type ImplicitProps e = [ImplicitProp]
6 data EntityDef e = EDef EName Props (ImplicitProps e) [FDecl e] [
  VTuple e]
```

EntityDef is a datatype giving function definitions. It has five arguments: the *name* of the entity type it defines, a list of property names and types, a list of *implicit* properties, which can be easily extended, but so far covers only the Id property, and finally a list of the entity's methods and a list of validation tuples related to that entity(??stx:vtuple)).

B.3.3 Templates

```
1 type TName = String
2 data TemplateDef t a = TDef TName [(PName, Type)] t
```

TemplateDef datatype give templates – callable, reusable components, which are equivalent of functions for the templates' component.

B.3.4 Pages

```
1 type PgName = String
2 data PageDef t a = PDef PgName [(PName, Type)] t
```

PageDef are a special type of template definitions, which give a full HTML page structure and cannot be embedded in other elements.

B.3.5 Program

```
1 type RequestParams = [(String, String)]
2
3 data Program e g f
4   = Program [e] (Maybe g)
5   | Fragment [e] (Maybe g) f
6   | Request [e] (Maybe g) (f, RequestParams)
7   | Sequence [e] (Maybe g) [(f, RequestParams)]
```

Program is a top level structure for storing WebDSL program syntax for various subsets of the language. It has three parameters: the type of definitions *e*, the type of global variables *g* and the type of entry point *f*. The first constructor, `Program`, has no entry point, and renders the main (root) page of the program. This constructor requires template components to be in scope. The second constructor, `Fragment`, gives a program with an arbitrary entry point, defining a fragment of a correct website. `Request` defines a program with entry point and a request parameter, which invokes the multi-phase processing mode. Finally, `Sequence` defines a server with multiple consecutive HTTP requests to be processed one after another.

The `Program` datatype is used at the top level binding function and in the test definitions.