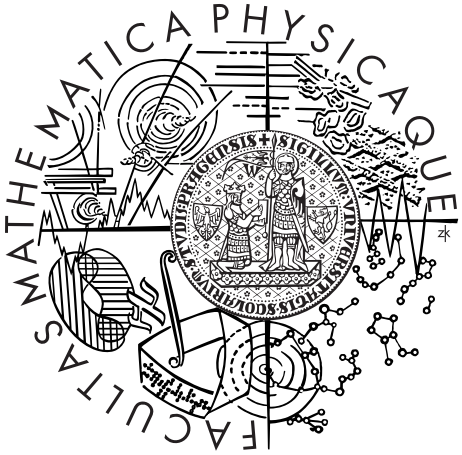
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Roman Smrž

Functional reactive programming for web applications

Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce:	RNDr. Petr Pudlák, Ph.D.
Studijní program:	informatika
Studijní obor:	obecná informatika

Praha 2011

Děkuji vedoucímu za zajímavý námět k práci a připomínky k výsledné podobě textu, díky nimž je snad o něco srozumitelnější. Dále chci poděkovat všem ze svého okolí, kteří mě nechali v klidu pracovat, a taky těm, již se zasloužili o to, abych nebyl do problematiky ponořen přespříliš hluboko nebo dlouho.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Název práce: Funkcionálně reaktivní programování pro webové aplikace

Autor: Roman Smrž

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: RNDr. Petr Pudlák, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Funkcionálně reaktivní programování umožňuje popis dynamických systémů deklarativním stylem s využitím typové bezpečnosti, což je doména obecně funkcionálních jazyků, zejména pak Haksellu, který jsme zvolili pro implementaci knihovny. Zde zkoumáme cesty, jimiž lze těchto technik využít při programování webových aplikací; konkrétně je zde navržen *domain specific language*, sloužící ke psaní webových stránek jako součást programu napsaného v Haskellu, který nakonec vygeneruje kód určený k odeslání uživateli, a k jejich obohacení o dynamický obsah. Také zjišťujeme, do jaké míry je možné rozšířit vyjadřovací sílu takové knihovny vzhledem k určitým omezením, která jsou daná zvoleným přístupem. Zároveň využijeme i stromovou strukturu HTML stránky, která se vcelku hodí k zápisu přímo v programovacím jazyce a do níž přidáváme další prvky dodávající systému dynamiku a interaktivitu.

Klíčová slova: Funkcionálně reaktivní programování, Haskell, Web, JavaScript

Title: Functional reactive programming for web applications

Author: Roman Smrž

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Petr Pudlák, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Functional reactive programming provides mechanisms of describing dynamic systems in a declarative and type-safe manner, which is traditional domain of functional languages in general and Haskell, which we use here, in particular. We explore ways, in which it may be applied to programming web applications, specifically, we design a domain specific language intended for writing web pages as a part of a Haskell program, which would generate the actual code served to the client, and enriching them with dynamic content; and also find out to what extent we may broaden the expressive power of such library given certain inherent constraints of the chosen method. In doing so, we utilize the tree-like structure of the HTML page, which is quite suitable to be written as a source code in a programming language and to which we embed additional entities providing the dynamics and interactivity.

Keywords: Functional reactive programming, Haskell, Web, JavaScript

Contents

Introduction

Functional reactive programming (FRP) is a general framework for composing dynamic systems in declarative way in pure functional languages, typically in Haskell. It introduces concepts of *behaviours*, which are values depending on time, and *events*, entities occurring at a point in time and carrying a value; those are then composed into a reactive system.

FRP has seen several implementations [2], differing in the intended use, interface details or types and levels of optimization. Those, however, deal with the case when the whole program is written and executed as Haskell code; in our case, the program is written in Haskell, providing the type safety and syntax possibilities of the language, and then compiled using ordinary Haskell compiler to a program that generates the JavaScript code implementing the functionality.

In this work, design of a domain specific language (DSL) given along with a proof-of-concept implementation for Haskell allowing programmer to implement client-side functionality (served in the form of HTML and JavaScript) of a web application using declarative style, which can be combined with the server-side part (if that is developed in Haskell as well, using frameworks such as Happstack [16]) in the same project or even the same module. Haskell-written server-side is not a requirement for this library (the code may be executed once to generate the HTML and JavaScript files and then any way of serving files and JSON-formatted data can be used), but may be useful nonetheless.

Since we are translating DSL structure into JavaScript at run-time, we deal with a system with limited set of functions; limited in the sense that we can not take arbitrary Haskell function and turn in into a JavaScript code (we would need a special Haskell-to-JavaScript compiler for that), but the system is Turing-complete, so any algorithm may be written using the pre-defined or user-defined data structures and functions manipulating with them. This, however, means that we can not use the usual notion of behaviours forming a monad (in $\mathcal{Hask}$), because we are not able to implement even the *return* function (we are not able to generally represent arbitrary Haskell values in JavaScript). Such structure forms a category where objects are types and morphisms are certain functions, similarly to the case of Haskell itself (disregarding the *seq* function); this category and it's properties shall be described later.

In the first chapter, the theoretical background of the whole system is given, mostly from the perspective of category theory. The practical aspect are described afterwards: the manipulation with data structures and composing programs in chapter 2 and construction of an HTML page along with handling of input and communication with a server in chapter 3. Overview of the implementation can be found in chapter 4 and some examples of use are given in chapter 5.

1. Theoretical foundation

In other FRP libraries, behaviours are usually to be both defined in Haskell and executed as Haskell expressions (that may be represented even directly as functions from time to a given type or using some more sophisticated definition [4, 5]). Here, however, we have expressions, which are defined in Haskell, but are evaluated into HTML and JavaScript code, which is then interpreted on the client side.

Most importantly, it means that we are not able to lift arbitrary values and functions to behaviours (it is possible, though, to lift arbitrary functions between behaviours, which will prove to be a very useful fact) and thus can not provide all the primitives found in other libraries, although those given here are sufficiently rich to be useful. It also means that we will not be interested that much in the semantics of Haskell expressions themselves, but rather that of the client-side code that they represent.

In this chapter we will develop formal semantics of this system and provide some category-theoretical definitions and properties of behaviours, which will differ from other libraries due to the limitations mentioned above. For some properties, we will use type classes from the package *categories* [9].

1.1 Defining behaviours

In the usual situation, behaviours form a monad. That means that if we denote B the type constructor of behaviours, then B together with the function $fmap :: \forall\alpha\,\beta.\,(\alpha \to b) \to (B\,\alpha \to B\,\beta)$ is a functor[1] and there exist functions

$$\begin{aligned} &return :: \forall\alpha.\,\alpha \to B\,\alpha \\ &join \quad :: \forall\alpha.\,B\,(B\,\alpha) \to B\,\alpha \end{aligned}$$

satisfying certain laws [1]. And every monad gives rise to a so-called Kleisli category, where objects remain the same (i.e. Haskell types), but morphisms between objects (types) α and β are now all the functions of type $\alpha \to B\,\beta$ [1]. These can be composed using Kleisli composition, in Haskell[2] implemented as an operator

$$\begin{aligned} &(\circ_m) \quad :: \forall\alpha\,\beta\,\gamma.\,(\beta \to m\,\gamma) \to (\alpha \to m\,\beta) \to (\alpha \to m\,\gamma) \\ &g \circ_m f = join \circ fmap\ g \circ f \end{aligned}$$

for any given monad m. It can be further shown, using monad laws, that this operator is associative and *return* is its both left and right identity, thus giving a composition of morphisms in a category. It is useful to note that such category may serve as a basis of definition instead of the original monad; in fact, given such category (along with the functor B), we may easily reconstruct the monad—*return* is just the identity in Kleisli category and $join = id_{B\,\alpha} \circ_B id_{B\,(B\,\alpha)}$ where id_* are identity functions in the original category.

[1] This means that $fmap\ id_\alpha = id_{B\alpha}$ and $fmap(g \circ f) = fmap\ g \circ fmap\ f$ for any f and g of compatible types.

[2] Actually, the name of the operator in Haskell is `<=<` and is to be found in the module Control.Monad, but for typographical reasons, $\circ_m$ is used here.

In the case of behaviours, morphisms in Kleisli category are semantically functions of type[3] $\alpha \to (\mathbb{T} \to \beta)$ or, equivalently, $\mathbb{T} \to \alpha \to \beta$; we shall denote this type as $BhvFun\,\alpha\,\beta$ for *behaviour functions*—functions from α to β which moreover depend on time. This is similar to signal functions in Arrowized FRP [7, 11], but here we do not deal with functions between behaviours/signals per se.

As we proposed above, definition of such functions together with their composition may be used as the basis of definition of behaviours; and if equipped with sufficient other structure (mostly the fact that $\mathbb{T} \to -$ is actually a functor) gives a monad of behaviours. That will not be our case, but this starting point gives us a way to express as much structure as we can in the terms of category-theoretical properties, which will be developed in the sections below.

1.2 Semantics

The semantics given here are for the resulting page which is to be provided to the client. However, if we treat the representation of a page in Haskell as some opaque value, which may be converted to a textual representation only through IO actions, then things, which will be semantically equal as per definitions here, may be considered equal even as expressions in Haskell, even though their representation may differ. This shall be used to justify some instances of certain type classes.

We will use the notation $[\![a]\!]$ to express semantic meaning of expression a. Given a domain D we will write $D_\perp$ to denote a *lifted* domain, id est one containing an extra $\perp$ (bottom, undefined) element.

The symbol $\mathbb{T}$ will stand for the domain of time, the only requirement of which being that it is totally ordered (in our library implemented using discrete sampling); and we introduce new type $Time$ with $[\![Time]\!] = \mathbb{T}$. The expression $a \to b$ will be overloaded for a) the function type from a to b and b) the semantic domain of all functions from a to b.

Most of our domains would be standard and lifted and not require deeper explanation. Of note are those of pair and function types, which will not be lifted:

$$[\![(\alpha,\ \beta)]\!] = [\![\alpha]\!] \times [\![\beta]\!]$$
$$[\![\alpha \to \beta]\!] = [\![\alpha]\!] \to [\![\beta]\!]$$

and instead we identify the bottom elements with already existing ones:

$$\perp = (\perp,\ \perp)$$
$$\perp = \lambda\, x.\, \perp$$

so that even though the run-time representations of those may differ, we will not provide any way to distinguish them.

[3] $\mathbb{T}$ denotes the domain of time; furthermore, some details concerning domains are omitted here and will be made more precise in the next section.

1.2.1 Behaviours

The goal is to have more or less typical FRP semantics even in our framework. There are different approaches among FRP implementations [2] and although we aim for the classical one [4], we will start (as noted in previous section) with the type constructor *BhvFun* with the semantics:

$$[\![BhvFun\ a\ b]\!] = \mathbb{T} \to [\![a]\!] \to [\![b]\!]$$

Next we define several basic operators:

For arbitrary type τ let $id^{Bhv}_{\tau} :: BhvFun\ \tau\ \tau$ with

$$[\![id^{Bhv}_{\tau}]\!](t) = \lambda x.\ x$$

For $f :: BhvFun\ \alpha\ \beta$ and $g :: BhvFun\ \beta\ \gamma$ let $g \circ_{Bhv} f :: BhvFun\ \alpha\ \gamma$ with

$$[\![g \circ_{Bhv} f]\!](t) = [\![g]\!](t) \circ [\![f]\!](t)$$

For $f :: BhvFun\ \alpha\ \beta$ and $g :: BhvFun\ \alpha\ \gamma$ let be $g \mathbin{\&\&\&} f :: BhvFun\ \alpha\ (\beta, \gamma)$ with semantics

$$[\![g \mathbin{\&\&\&} f]\!](t) = \lambda x.\ ([\![g]\!](t)(x),\ [\![f]\!](t)(x))$$

Next, we will define couple of behaviour functions which do not depend on time. For arbitrary types τ and σ:

$$\begin{array}{ll} fst :: BhvFun\ (\tau,\ \sigma)\ \tau & [\![fst]\!](t) = \lambda(x,\ y).\ x \\ snd :: BhvFun\ (\tau,\ \sigma)\ \sigma & [\![snd]\!](t) = \lambda(x,\ y).\ y \end{array}$$

Finally, for a type *Void*[4] with $[\![Void]\!] = \emptyset_{\perp} = \{\perp\}$, we define

$$\textbf{type}\ Bhv\ a = BhvFun\ Void\ a$$

thus getting the type of behaviours with the semantics we originally intended:

$$[\![Bhv\ \alpha]\!] = [\![BhvFun\ Void\ \alpha]\!] = \mathbb{T} \to \{\perp\} \to [\![\alpha]\!] \cong \mathbb{T} \to [\![\alpha]\!]$$

and we will use the last form as the semantics of $Bhv\ \alpha$.

1.2.2 Events

Combining behaviours together with events provides the reactivity of the system. But before we get to the interaction of those two, we need to first define the latter. As opposed to the behaviours, which are time-varying values, events occur at a point in time while carrying some value of given type. So, they can be defined as pairs of time and value [4] or, for practical reasons, as lists of time-value pairs, non-decreasing in the time component [5].

[4] In, Haskell, the type *unit*, denoted (), is usually used in cases, where some "empty" type is required; it the world with bottoms, however, this type has actually two values: () and $\perp_{()}$, and as such is not suitable here.

In our case, we will use the second definition, thus[5]

$$[\![Event\ a]\!] = [(\mathbb{T} \times [\![a]\!])]$$

With the difference that we will require the time to be strictly increasing.

We will, however, not allow working directly with such list (nor actually use it in representation). Instead, the most general interface for events that we will provide will be *evfold*, which does a left fold[6] over all the values of given event up to the current time; formally:

$$\begin{aligned}
&evfold :: (Bhv\ Time \to Bhv\ \alpha \to Bhv\ \beta \to Bhv\ \alpha) \to \\
&\qquad Bhv\ \alpha \to Event\ \beta \to Bhv\ \alpha \\
&[\![evfold\ f\ x\ e]\!](t) = fold\ [\![f]\!]\ [\![x]\!]\ [(t', y) \mid (t', y) \in [\![e]\!],\ t' \leq t] \\
&\quad \textbf{where}\ fold\ \overline{f}\ \overline{x}\ [] = \overline{x} \\
&\qquad fold\ \overline{f}\ \overline{x}\ ((t, y) : r) = fold\ \overline{f}\ (\overline{f}\ t\ \overline{x}\ y)\ r
\end{aligned}$$

In our framework, we will represent the events not as list of occurrences, but rather as behaviours carrying the information of the last instance of the event in any given time (or special value indicating that no event occurred yet). This is the reason, why we required the times to be strictly increasing (for in such case, those lists exactly correspond to our representation). For this purpose, we introduce new data type:

$$\begin{aligned}
&\textbf{data}\ Timed\ \alpha = NotYet \\
&\qquad\qquad \mid\ OnTime\ Time\ \alpha \\
&\textbf{type}\ Event\ \alpha = Bhv\ (Timed\ \alpha)
\end{aligned}$$

This allows us to use all the functions and combinators both developed so far and introduced later for behaviours to also manipulate events. Specifically, in the section 2.2, we will describe means of manipulating various data structures, including *Timed*. That will suffice to provide definitions of never-occurring event or functions like

$$evmap :: (Bhv\ \alpha \to Bhv\ \beta) \to Event\ \alpha \to Event\ \beta$$

mapping some function on event and keeping times,

$$\begin{aligned}
&evmerge :: (Bhv\ \alpha \to Bhv\ \alpha \to Bhv\ \alpha) \to \\
&\qquad Event\ \alpha \to Event\ \alpha \to Event\ \alpha
\end{aligned}$$

merging two events (and using given function for values of occurrences coinciding in the same time), or

$$switcher :: Bhv\ \alpha \to Event\ (Bhv\ \alpha) \to Bhv\ \alpha$$

where b_0 ‘*switcher*‘ e behaves initially like b_0 and changes on each occurrence of e to the behaviour provided by it.

Such definitions will be straightforward and will not be given here. They are, nevertheless, available in the source code of the library itself.

[5]For brevity, we will use the Haskell notation for lists in this section even for semantic expressions, but the meaning should be clear.

[6]Compare with standard $foldl :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$

1.3 The category of behaviours

We started all our definitions with the idea of certain category whose morphism were time-varying functions, we will call this category $\mathcal{Bhv}$. Objects of $\mathcal{Bhv}$ are Haskell types and morphism between α and β are all functions from the domain $\mathbb{T} \to [\![\alpha]\!] \to [\![\beta]\!]$, represented as expressions of type $BhvFun\ \alpha\ \beta$.

Composition of morphisms is given by the function $\circ_{Bhv}$ and identity on object α is the id_α^{Bhv}. For this to form a category, we need to verify that the identity really is an identity:

$$[\![id^{Bhv} \circ_{Bhv} f]\!] = \lambda t.\, (\lambda x.\, x) \circ [\![f]\!](t) = \lambda t.\, [\![f]\!](t) = [\![f]\!]$$

(and similarly for the other case) and that the composition is associative, which follows simply as well:

$$[\![(h \circ_{Bhv} g) \circ_{Bhv} f]\!] = \lambda t.\, ([\![h]\!](t) \circ [\![g]\!](t)) \circ [\![f]\!](t) =$$
$$= \lambda t.\, [\![h]\!](t) \circ ([\![g]\!](t) \circ [\![f]\!](t)) = [\![h \circ_{Bhv} (g \circ_{Bhv} f)]\!]$$

Void is apparently terminal object in $\mathcal{Bhv}$ since it behaves similarly to a singleton in the category $\mathcal{Set}$. Likewise, for any two objects α and β of $\mathcal{Bhv}$, there exists object (α, β), which, since we decided to use unlifted tuples, is in fact their product in a categorical sense. Thus, the category $\mathcal{Bhv}$ has all finite products.

We also already described semantics of the types of the form $\alpha \to \beta$; if we fix some type α, we may construct an endofunctor in $\mathcal{Bhv}$ denoted $\alpha \to -$ in the following way: for an object (type) β, $(\alpha \to -)\beta = \alpha \to \beta$ and for a morphisms m in $\mathcal{Bhv}$ let $[\![(\alpha \to -)m]\!](t) = \lambda f.\, [\![m]\!](t) \circ f$.

We want to show that this functor is the right adjoint for the functor $(-, \alpha)$ (constructed in a similar way), which (since $\mathcal{Bhv}$ is locally small) can be formulated [10] as the existence of isomorphism

$$\eta : [\![BhvFun\ (\tau, \alpha)\ \sigma]\!] \to [\![BhvFun\ \tau\ (\alpha \to \sigma)]\!]$$
$$\eta(f) = \lambda t.\, \lambda x.\, \lambda y.\, f(t)(x, y)$$

and the fact that this isomorphism is natural in both τ and σ; we shall write *curry* for behaviour function with $[\![curry]\!] = \eta$ and $[\![uncurry]\!] = \eta^{-1}$.

A category with all these properties (having terminal object, finite products and just described adjunction) is said to be *cartesian closed* [1, 10]. This also uniquely determines an "application" function, which we will define explicitly:

$$apply :: BhvFun\ ((\tau \to \sigma), \tau)\ \sigma$$
$$[\![apply]\!](t) = \lambda(f, x).\, f\ x$$

Here, we should make explicit certain difficulties arising from the fact that expressions of behaviours exist in both the JavaScript and Haskell worlds. As stated in the beginning, the semantics described here concern the JavaScript part. However, the behaviours, mainly the type *BhvFun*, is also represented in Haskell, for which we made instances of various categorical type classes.

However, the proper semantics of those expressions in Haskell must include one extra Haskell-level bottom and this includes all the objects and morphism

in our category. Although even with such modification, it does not cease to be a category (since our composition is strict on the Haskell level), it loses many of the properties described here, like products or exponentials; thus, such "more lifted" category is no longer cartesian closed.

Yet note that all those instances, although not strictly correct, are used rather for convenience and definitely not crucial to working of our library.

1.4 Embedding $\mathcal{Bhv}$ into $\mathcal{Hask}$

Such category of behaviours as hitherto defined is however not a subcategory of $\mathcal{Hask}$, it is represented as an instance of a Category type class. This is not ideal, because although we can use this definition and compose basic functions into useful programs using composition and combinators like &&&, this way of programming gets quickly rather cumbersome, especially when dealing with multi-parameter functions and alike. To solve this problem, the approach of using *arrows* [8] emerged and along with *arrow notation* [12] allows rather pleasant way of writing programs. Neither this would help us though, because we can not provide the *arr* (sometimes called *pure*) combinator required for the *Arrow* type class instance.

We constructed a cartesian closed category which as such may be considered equivalent to some typed λ-calculus [10]. In this situation we may expect to somehow elevate the relevant part of Haskell syntax for our expressions. Although we will not get to use all of the language features (most notably we will have to do without pattern matching), we will in this section show, how to write rather naturally-looking Haskell code for the behaviour function we deal with here.

We shall create a bijective mapping between expressions of type $BhvFun\ \alpha\ \beta$ and those of type $Bhv\ \alpha\ \rightarrow\ Bhv\ \beta$, thus defining an embedding of $\mathcal{Bhv}$ on a full subcategory of $\mathcal{Hask}$ generated by the types of the form $Bhv\ \alpha$.

One of the directions is rather easy: using the composition operator

$$(\circ_{Bhv})\ ::\ \forall\tau\ \sigma\ \rho.\ BhvFun\ \sigma\ \rho\ \rightarrow\ BhvFun\ \tau\ \sigma\ \rightarrow\ BhvFun\ \tau\ \rho$$

if we instantiate $\sigma\ =\ \alpha$, $\rho\ =\ \beta$ and $\tau\ =\ Void$ we get

$$\begin{aligned} bhvToHask = (\circ_{Bhv})\ ::\ & \\ & BhvFun\ \alpha\ \beta\ \rightarrow\ BhvFun\ Void\ \alpha\ \rightarrow\ BhvFun\ Void\ \beta\ = \\ & \quad =\ BhvFun\ \alpha\ \beta\ \rightarrow\ (Bhv\ \alpha\ \rightarrow\ Bhv\ \beta) \end{aligned}$$

The harder part is the other direction. Although we have the operator *apply*, which will be required, we will also need to somehow represent functions of type $Bhv\ \alpha\ \rightarrow\ Bhv\ \beta$ in our run-time JavaScript part; in other words, we need some function

$$\begin{aligned} & cb\ ::\ \forall\alpha\ \beta.\ (Bhv\ \alpha\ \rightarrow\ Bhv\ \beta)\ \rightarrow\ Bhv\ (Bhv\ \alpha\ \rightarrow\ Bhv\ \beta) \\ & [\![cb\ f]\!](t)\ =\ [\![f]\!] \end{aligned}$$

It will be implemented basically by creating JavaScript function, which as a parameter gets a behaviour that is formally passed to the Haskell function f

and the representation of the result is included in the body of the JavaScript function. Whole process will be described in more detail in chapter 4, dedicated to implementation, and naturally in the source code of the library.

The last instruments required are two operators *bhvWrap* and *bhvUnwrap* with apparent semantics.

$$\begin{array}{ll} bhvWrap \;::\; \forall\alpha.\; BhvFun\;\alpha\;(Bhv\;\alpha) & [\![bhvWrap]\!](t) \;=\; \lambda x.\;(\lambda t'.\;x) \\ bhvUnwrap \;::\; \forall\alpha.\; BhvFun\;(Bhv\;\alpha)\;\alpha & [\![bhvUnwrap]\!](t) \;=\; \lambda f.\;f(t) \end{array}$$

With all these, we are finally able to define

$$\begin{array}{l} haskToBhv \;::\; \forall\,\alpha\,\beta.\;(Bhv\;\alpha \;\rightarrow\; Bhv\;\beta) \;\rightarrow\; BhvFun\;\alpha\;\beta \\ haskToBhv\;f \;=\; bhvUnwrap\;\circ_{Bhv}\;apply\;\circ_{Bhv}\;(cb\;f\;\&\&\&\;bhvWrap) \end{array}$$

giving the semantics

$$\begin{array}{l} [\![haskToBhv\;f]\!] = \\ = [\![bhvUnwrap\;\circ_{Bhv}\;apply\;\circ_{Bhv}\;(cb\;f\;\&\&\&\;bhvWrap)]\!] = \\ = \lambda t.\;[\![bhvUnwrap]\!](t) \circ [\![apply]\!](t) \circ [\![cb\;f\;\&\&\&\;bhvWrap]\!](t) = \\ = \lambda t.\;(\lambda g.\;g\;t) \circ (\lambda(h,\;y).\;h(y)) \circ (\lambda x.\;([\![f]\!],\;\lambda t'.\;x)) = \\ = \lambda t.\;\lambda x.\;(\lambda g.\;g\;t)\;(\;(\lambda(h,\;y).\;h(y))\;([\![f]\!],\;\lambda t'.\;x)\;) = \\ = \lambda t.\;\lambda x.\;(\lambda g.\;g\;t)\;([\![f]\!](\lambda t'.\;x)) = \\ = \lambda t.\;\lambda x.\;[\![f]\!]\;(\lambda t'.\;x)\;t \end{array}$$

Which is a behaviours function that passes its parameter to the function f as a constant (not depended on time) behaviour and returns the result. As such, it is a left inverse of the function *bhvToHask* defined earlier:

$$[\![bhvToHask\;f]\!] \;= \lambda b.\;\lambda t.\;[\![f]\!]\;t\;(b\;t)$$

$$\begin{array}{l} [\![haskToBhv\;(bhvToHask\;f)]\!] = \\ = \lambda t.\;\lambda x.\;((\lambda b.\;\lambda t.\;[\![f]\!]\;t\;(b\;t))(\lambda t'.\;x))\;t = \\ = \lambda t.\;\lambda x.\;(\lambda t.\;[\![f]\!]\;t\;((\lambda t'.\;x)\;t))\;t = \\ = \lambda t.\;\lambda x.\;[\![f]\!]\;t\;x = [\![f]\!] \end{array}$$

Due to the fact that the behaviour passed as a parameter to the f in the implementation of $haskToBhv\;f$ discards its time parameter, it is not exactly right inverse; the expression we get is:

$$\begin{array}{l} [\![bhvToHask\;(haskToBhv\;f)]\!] = \\ = \lambda b.\;\lambda t.\;(\lambda t.\;\lambda x.\;([\![f]\!](\lambda t'.\;x))\;t)\;t\;(b\;t) = \\ = \lambda b.\;\lambda t.\;[\![f]\!]\;(\lambda t'.\;b\;t)\;t \end{array}$$

But since we are concerned only in evaluating all the behaviours at the same point in time, it does not make a difference to us. Thus, we can freely convert between these two representations of behaviour functions. The form of Haskell function between behaviours will be useful for writing expressions in Haskell; the other one will be used for converting all the code into JavaScript and running on the client side, where the only thing we need to implement are the primitives dealing with the basic definition of behaviour functions from which we started.

2. Working with behaviours

In the previous chapter, we laid a theoretical foundation for our framework of behaviours and behaviour functions. From morphisms in certain category, we got to the point when we can work with almost ordinary Haskell functions, just with somewhat modified types: all the types apart from the function one will be "wrapped" in the type constructor *Bhv*. So, instead of function

$$map :: (\alpha \to \beta) \to [\alpha] \to [\beta]$$

we will have

$$map_b :: (Bhv\ \alpha \to Bhv\ \beta) \to Bhv\ [\alpha] \to Bhv\ [\beta]$$

This allows us to use Haskell syntax for lambda abstraction and application and still be able to take these expressions and turn them into client-side JavaScript code. Here, we shall use the subscript '*b*' for functions with the modified type provided by our library, although no such differentiation is used in the library itself (which is nevertheless possible using module qualification).

Here, we will describe, how to construct and work with these function, manipulate data structures and make useful programming constructs. Since we also use the lazy evaluation strategy for the client-side code, the strictness characteristics of the functions presented here are the same as of their counterparts in the Haskell itself.

2.1 Values into behaviours

As we originally said, we are not able to represent arbitrary Haskell value as a behaviour. Nothing stops us, however, from representing a chosen subset of those values (selected by a type class) and turning them into constant behaviours. We will use a function *cb* (constant behaviour), which we already encountered for a special case of functions between behaviours. In general, its type is

$$cb :: \forall \alpha\ \beta.\ BhvValue\ \alpha \Rightarrow \alpha \to BhvFun\ \beta\ \alpha$$

Often used in the specialized version as

$$cb :: \forall \alpha.\ BhvValue\ \alpha \Rightarrow \alpha \to Bhv\ \alpha$$

The type class *BhvValue* contains a method that describes how to represent given value in the JavaScript code. Instances are provided for the standard Prelude types, the types $BhvFun\ \alpha\ \beta$, $Bhv\ \alpha \to Bhv\ \beta$ and can be added for new ones; we will however omit the details for now.

If this is sort of like a *return* function from the monad definition (just restricted to types in a certain type class), we may also introduce the equivalent of the *join*:

$$bjoin :: \forall \alpha.\ Bhv\ (Bhv\ \alpha) \to Bhv\ \alpha$$

Note that this one works for arbitrary type, without any type class constraints. Further, for reasons, which will became clear later, we will use a slightly more general version

$$bjoin :: \forall \alpha.\ Bhv\ (BhvFun\ \alpha\ \beta) \to BhvFun\ \alpha\ \beta$$

2.2 Manipulating data structures

We have already developed some tools to combine and manipulate behaviour functions themselves. In a real world, we will however need to also work with various data represented as data structures like tuples, list and others; now it is time to show how to deal with them. Specifically, we will describe how to work with algebraic data types (ADT); other kinds of data will not be treated here, but for some, similar approach may work.

If we want to create a value of some algebraic data type in Haskell, the most straightforward way to do so is using constructors of that type. The same way is used in our library. The only difference is, in order to be able to work with behaviours, they are used in the form introduced in the previous section. For example, to construct lists, we are given[1]:

$$
\begin{aligned}
&\mathit{nil}_b \;::\; \forall\alpha.\; \mathit{Bhv}\;[\alpha] \\
&\mathit{cons}_b \;::\; \forall\alpha.\; \mathit{Bhv}\;\alpha \;\to\; \mathit{Bhv}\;[\alpha] \;\to\; \mathit{Bhv}\;[\alpha]
\end{aligned}
$$

The first function of the two returns an empty list (more precisely, a behaviour, which is constantly an empty list). The second one gets some behaviour of type α and constructs a list of such values (behaviour thereof, that is) by prepending it to its second argument. Similar constructors can be made for any other ADT.

The other thing we need to do is take some value, look what it is and decide what to do with its contents. In Haskell, this is usually taken care of using *pattern matching*, which is very useful and elegant language feature. As already noted, we will not be able to use that, so instead we shall use functions, which may be called *destructors* (as they are complementary to constructors).

Those are functions, which, apart from the value they are about to "destruct", take also for each of the possible constructors a function determining what to do with a value constructed by it. For example, if lists can be constructed either by nil_b from no value at all, or by cons_b from two values of types $\mathit{Bhv}\;\alpha$ and $\mathit{Bhv}\;[\alpha]$, then the destructor for list is

$$
\begin{aligned}
&\mathit{list}_b \;::\; \mathit{Bhv}\;\beta && \text{— what to do with empty list} \\
&\quad\to\; (\mathit{Bhv}\;\alpha \;\to\; \mathit{Bhv}\;[\alpha] \;\to\; \mathit{Bhv}\;\beta) && \text{— what to do with head and tail} \\
&\quad\to\; \mathit{Bhv}\;[\alpha] && \text{— the list to destruct} \\
&\quad\to\; \mathit{Bhv}\;\beta && \text{— the resulting behaviour}
\end{aligned}
$$

So we can write the equivalent of the function *drop* 1 (more interesting examples can be given once we show how to use recursion in the next section).

$$\mathit{dropOne} \;=\; \mathit{list}_b\;\mathit{nil}_b\;(\lambda\;\_\;t \;\to\; t)$$

But such destructors are in several cases to be find even in the standard Haskell in the form of functions

$$\mathit{maybe} \;::\; \beta \;\to\; (\alpha \;\to\; \beta) \;\to\; \mathit{Maybe}\;\alpha \;\to\; \beta$$

[1] In our library, we use the operator $\sim$: instead of *cons*; it is similar to the standard :, which we can not use, because colon-prefixed operators are reserved for constructors and our constructors are, from the point of view of Haskell, just ordinary functions.

$$either \;::\; (\alpha \;\to\; \gamma) \;\to\; (\beta \;\to\; \gamma) \;\to\; Either\;\alpha\;\beta \;\to \gamma$$

These two are of course provided also in our library, just modified to work with behaviours as usual. To stay consistent with these, we use the name of the type as the name of destructor and the destructed value is always passed as the last argument.

To enable the use of new data structures in behaviours, it is necessary to provide the constructors and destructor. These currently need to be implemented as JavaScript primitives and are provided for the types defined in the standard Haskell Prelude.

Since all the algebraic data types are similar in the structure, it would be possible to provide some generic way of adding their support; possibly in a way using a system like TemplateHaskell [13], where the user would just describe the number of constructors and their argument types and got all the interface functions defined. This is, however, more of a technical issue and we will not deal with it here.

2.3 Recursion and fixpoint operator

To allow not only peeking at the first few layers of some structure, but also traversing it fully and generally make the system of behaviours Turing-complete, we need to be able to write recursive functions. Using recursion directly in Haskell does not work, because it generates the recursive tree in the execution of Haskell code and that can not be compiled into the client-side JavaScript (since that code has to be finite). Instead, as is usual in the lambda calculus, we will provide a fixpoint operator.

The basic function we will work with is a modified version of standard *fix*

$$fix_b \;::\; (Bhv\;\alpha \;\to\; Bhv\;\alpha) \;\to\; Bhv\;\alpha$$

returning a behaviour of the least fixed point of the function given as the first parameter. More formally, if $haskToBhv\,f = \bot$ (on the Haskell level), then $fix_b\,f = \bot$ as well, otherwise $[\![fix_b\,f]\!](t)$ is the least fixed point of the function $[\![haskToBhv\,f]\!](t)$; this is achieved the same way it is in Haskell using lazy evaluation, which, implementing the leftmost evaluation strategy, is guaranteed to reach a normal form, if that exists [3].

So, if we wanted to write for example the $length_b$ function, we can use the fact that it is a fixed point of the

$$\begin{aligned} &l \;=\; \lambda f \;\to\; list_b\;0\;(\lambda\_\;xs \;\to\; 1 \;+_b\; f\;xs) \\ &::\; (Bhv\;[\alpha] \;\to\; Bhv\;Int) \;\to\; (Bhv\;[\alpha] \;\to\; Bhv\;Int) \end{aligned}$$

This expression is however not of the form $Bhv\;\alpha \;\to\; Bhv\;\alpha$ we used for the operator fix_b. To overcome this problem, we may first use the functions *haskToBhv* and *bhvToHask* getting

$$haskToBhv\;.\;l\;.\;bhvToHask \;::\; BhvFun\;[\alpha]\;Int \;\to\; BhvFun\;[\alpha]\;Int$$

Which can be further altered to

$$cb\ .\ haskToBhv\ .\ l\ .\ bhvToHask\ .\ bjoin$$
$$::\ Bhv\ (BhvFun\ [\alpha]\ Int)\ \rightarrow\ Bhv\ (BhvFun\ [\alpha]\ Int)$$

And this expression can be finally passed to the fix_b giving the desired function

$$\begin{aligned} length_b\ =\ fix_b(&cb\ .\ haskToBhv\ .\ (\lambda f\ \rightarrow\ list_b\ 0\ (\lambda\_\ xs\ \rightarrow\ 1\ +_b\ f\ xs)) \\ &.\ bhvToHask\ .\ bjoin) \\ &::\ Bhv\ (BhvFun\ [\alpha]\ Int) \end{aligned}$$

And that can be further transformed back into the expression of the originally desired type $Bhv\ [\alpha]\ \rightarrow\ Bhv\ Int$ using additional calls to *bhvToHask* and *bjoin*.

In the case of functions having multiple arguments we have to use additional helper functions

$$\begin{aligned} &curry_b\ ::\ (Bhv\ (\alpha,\ \beta)\ \rightarrow\ \gamma)\ \rightarrow\ Bhv\ \alpha\ \rightarrow\ Bhv\ \beta\ \rightarrow\ \gamma \\ &curry_b\ f\ x\ y\ =\ f\ (x\ \&\&\&\ y) \\ &uncurry_b\ ::\ (Bhv\ \alpha\ \rightarrow\ Bhv\ \beta\ \rightarrow\ \gamma)\ \rightarrow\ Bhv\ (\alpha,\ \beta)\ \rightarrow\ \gamma \\ &uncurry_b\ f\ x\ =\ f\ (fst\ .\ x)\ (snd\ .\ x) \end{aligned}$$

where the γ is meant to be of the form $Bhv\ \gamma_1\ \rightarrow\ \cdots\ \rightarrow\ Bhv\ \gamma_n$. To avoid using loads of calls to *curry* and *uncurry*, we will use more general functions

$$\begin{aligned} curryAll_b\ ::\ &Bhv\ (\alpha_1,\ (\alpha_2,\ (\ldots\ (\alpha_{n-1},\ \alpha_n)\ \ldots))\ \rightarrow \\ &(Bhv\ \alpha_1\ \rightarrow\ Bhv\ \alpha_2\ \rightarrow\ \cdots\ \rightarrow\ Bhv\ \alpha_{n-1}\ \rightarrow\ Bhv\ \alpha_n)) \\ uncurryAll_b\ ::\ &(Bhv\ \alpha_1\ \rightarrow\ Bhv\ \alpha_2\ \rightarrow\ \cdots\ \rightarrow\ Bhv\ \alpha_{n-1}\ \rightarrow\ Bhv\ \alpha_n)\ \rightarrow \\ &Bhv\ (\alpha_1,\ (\alpha_2,\ (\ldots\ (\alpha_{n-1},\ \alpha_n)\ \ldots))) \end{aligned}$$

Although the types given here are not directly legal in Haskell, they are clearer than the actual ones we get by rather straightforward definitions using type classes (some extensions to the standard Haskell are needed for that, though), where each step is provided by a call to *curry* or *uncurry*, respectively.

To make writing of recursive functions much easier and resulting code cleaner than what was used here in the case of $length_b$ (not to mention the case of dealing with currying), we shall provide a convenience function

$$\begin{aligned} bfix\ ::\ &((Bhv\ \alpha_1\ \rightarrow\ \cdots\ \rightarrow\ Bhv\ \alpha_n)\ \rightarrow\ (Bhv\ \alpha_1\ \rightarrow\ \cdots\ \rightarrow\ Bhv\ \alpha_n))\ \rightarrow \\ &(Bhv\ \alpha_1\ \rightarrow\ \cdots\ \rightarrow\ Bhv\ \alpha_n) \end{aligned}$$

that will do all the necessary plumbing. Its implementation is for the most part seen in the example with $length_b$ given above, just enriched of the additional calls to the above defined $curryAll_b$ and $uncurryAll_b$. With this functions, recursive definitions may be programmed in basically the same way as is possible in the Haskell with the function *fix*. To show the $length_b$ again:

$$\begin{aligned} &length_b\ ::\ Bhv\ [\alpha]\ \rightarrow\ Bhv\ Int \\ &length_b\ =\ bfix\ \$\ \lambda f\ \rightarrow\ list_b\ 0\ (\lambda\_\ xs\ \rightarrow\ 1\ +_b\ f\ xs) \end{aligned}$$

And the same also works for functions with more parameters:

$$\begin{aligned} &foldl_b\ ::\ (Bhv\ \alpha\ \rightarrow\ Bhv\ \beta\ \rightarrow\ Bhv\ \alpha)\ \rightarrow\ Bhv\ \alpha\ \rightarrow\ Bhv\ [\beta]\ \rightarrow\ Bhv\ \alpha \\ &foldl_b\ f\ =\ bfix\ \$\ \lambda fld\ \rightarrow\ \lambda z\ \rightarrow\ list\ z\ (\lambda x\ xs\ \rightarrow\ fld\ (f\ z\ x)\ xs) \end{aligned}$$

2.4 Going back to Haskell

It is nice that we can write expressions that may be turned into JavaScript code and run in the client's browser. Yet in some situation, it may be desirable to actually execute the code with behaviours directly in the Haskell program itself. For this purpose, we will show some "evaluation" functions; the basic primitive of those is

$$\mathit{unsafeBfEval} \;::\; \mathit{BhvFun}\;\alpha\;\beta \;\rightarrow\; (\alpha \;\rightarrow\; \beta)$$

The behaviour function is executed as if in the time at the beginning; no events occurred, no input was entered and so on. The *unsafe-* prefix is used because it loses the requirement that the value of type α is represented in JavaScript and thus, if we generate some values of type *Bhv*, those may fail if we try to pass them into the JavaScript world; the simplest example is the function

$$\mathit{unsafeBfEval}\;\mathit{bhvWrap} \;::\; \alpha \;\rightarrow\; \mathit{Bhv}\;\alpha$$

without the type class constraint on the type α seen in the function *cb*, so it can not work correctly. Similar issue arises when evaluating behaviours passed through *HtmlM* as described in the next chapter.

Since we usually do not work directly with behaviour functions, but rather with the more practical functions between behaviours, we need to also provide some "unwrapping" utility functions; defined again using a type class:

$$\begin{aligned}
&\textbf{class}\;\mathit{BhvEval}\;\alpha\;\beta \mid \alpha \;\rightarrow\; \beta\;\textbf{where}\\
&\quad \mathit{unsafeEval} \;::\; \alpha \;\rightarrow\; \beta\\
\\
&\textbf{instance}\;\mathit{BhvEval}\;(\mathit{Bhv}\;\alpha)\;\alpha\;\textbf{where}\;\ldots\\
\\
&\textbf{instance}\;(\mathit{BhvEval}\;\alpha\;\alpha',\; \mathit{BhvEval}\;\beta\;\beta') \;\Rightarrow\\
&\quad \mathit{BhvEval}\;(\alpha \;\rightarrow\; \beta)\;(\alpha' \;\rightarrow\; \beta')\;\textbf{where}\;\ldots
\end{aligned}$$

Now, we are able to write $(\mathit{unsafeEval}\;\mathit{length}_b)$ "abc" and get 3 as a result directly in the Haskell code; since the type of $\mathit{unsafeEval}\;\mathit{length}_b$, $\forall\alpha.\,[\alpha] \;\rightarrow\; \mathit{Int}$, does not contain any behaviours, it is equivalent to the standard function *length*.

3. Page construction

So far, we dealt with definition of behaviours and ways how to use them to construct expressions suitable to be send to and executed by the client web browser. Now we show how to produce some output in the form of HTML page which may be displayed to the user.

Here we will cover construction of basic (non interactive) page, followed by means how to perform client-side computation and communication with the server. Then will be presented means of constructing mutually dependent parts of a page using value recursion.

3.1 Basic page layout

For a static page, we use a system of combinators similar to the one used in the BlazeHtml library [14], those familiar with this library would not encounter any new concepts in this section. We will use a data type *Html* representing HTML snippets or even the whole page (its implementation will be described later). Each individual tag is constructed using alike-named function like *html*, *body*, *p*, *img* and so on.

Those combinators are of two kinds: one without parameters for void tags like $img :: Html$ or $br :: Html$; the other gets a parameter determining its contents in the cases of $body :: Html \to Html$ or $p :: Html \to Html$ and others.

Attributes are represented using data type *Attribute* and constructed again using functions named after them, which take a string parameter to be used as an attribute's value, for example with functions like $style :: String \to Attribute$ or $name :: String \to Attribute$. They may be added to tags using the (!) operator with construction like:

p ! *class_* "first" (img ! *src* "image.png")

Since (!) has to work with expressions of type *Html* (like *img* above) and with those of type $Html \to Html$ (like p), it is implemented using a type class:

class *Attributable a* **where** $(!) :: a \to Attribute \to a$
instance *Attributable Html* **where** ...
instance *Attributable* $(Html \to Html)$ **where** ...

In several cases, where the name of HTML tag or attribute clashes with some Haskell keyword or common function (like *type*, *class*, *head* or *div*), we will append underscore to the name of HTML-manipulation functions.

HTML structure is tree-like and although it would be possible to compose whole page given the expression above, it would result in fair number of parenthesis making the whole code less readable. Instead, we can use the *do notation* available in Haskell to mark nested blocks. To do this, we need a type that is an instance of the *Monad* type class, so the whole definition has following form:

```
data HtmlM a = ...
instance Monad Html where ...
type Html = HtmlM ()
```

where the *Monad* instance is written so that for two values, x and y, of type *Html* the expression $x \gg y$ is concatenation of those two. Then we may write code like this

```
page = html $ do
                head_ $ do
                        title "title"
                body $ do
                        p $ do
                                str "first"
                                img ! src "image1.png"
                        p $ do
                                str "second"
                                img ! src "image2.png"
```

Although unlike the BlazeHtml library, we will actually need the *HtlmM Monad* instance for maintaining some internal state needed for handling behaviours. Similary, for the purpose of using the *do* syntax, the type parameter of *HtmlM* could very well be just phantom, but we will use it later to extract certain behaviours and events from various input elements (textfields, buttons, forms et cetera).

3.2 The dynamic elements

The core functionality of our library comes with the possibility of incorporating non-static components into the web page. The basic primitive to achieve this is the function *bhv*—given some variable x of type *Bhv Html* we may embed it into the page with just "*bhv x*".

To generate such expression, one can either create ordinary *Html* value and turn it into constant behaviour using *cb* (and probably subject it to further manipulation first), or take a behaviour of one of the supported types and apply *toHtml* to it. Those types include *Int*, *String*, [*Html*] or *Maybe Html*.

So, for example suppose we have a variable *text* :: *Bhv String* and we want to display its length, than the following code will do just that (along with enclosing it in a span tag):

```
span_ $ do
        str "the length is: "
        bhv $ toHtml $ length_b x
```

3.2.1 Getting input

As well as important producing the output is, we also need to be able to get some information from the user. For such purposes is the HTML equipped with various input elements. We utilize them by using using functions like *textfield*, which not

only create the appropriate HTML snippet (`<input type="text">` in this case), but also return a behaviour representing the value of the input element.

To expand a little our last example so that it displays the length of the text currently written in the input field:

```
span_ $ do
        x ← textfield
        str "the length is: "
        bhv $ toHtml $ length_b x
```

Texfields represent their value as a behaviour of a string, for other elements, it makes more sense to provide events representing certain actions, like clicking on a button or sending a form. So we have functions like

```
button :: HtmlM (Event ())
form :: HtmlM (Event [(String, String)])
```

where the event returned by *button* triggers every time the button is clicked on and the one from *form* provides also the data entered to it when the form was submitted.

3.2.2 Talking to a server

To communicate with a server, we use the JSON format [17]. In order to further work with such values, we need to be able to deserialize them; to identify the types for which such functionality is implemented, we use the *BJSON* class (similar to the *JSON* in the package *json* [18] we actually use in the server part):

```
class BJSON α where
  readJSON_b :: Bhv JSValue → Bhv (Result α)
  writeJSON_b :: Bhv α → Bhv JSValue
```

Instances are again provided for the standard types and additional ones can be implemented using versions of the functions from the *json* package modified for the use of behaviours and provided by our library.

The simpler variant of communication is taking a simple value from the server using the GET request type; this is done by the function

```
sget :: BJSON α ⇒ String → Bhv (Maybe α)
```

The behaviour *sget* "name" is initially *Nothing* and once the browser gets a response from the server for the request `?q=name` and that response is successfully parsed, the behaviour changes to the appropriate *Just* value.

Due to the fact that it is not always possible to infer the type (depending on the subsequent use of the behaviour), it may be needed to specify the type explicitly:

```
span_ $ do
        let name = sget "name" :: Bhv (Maybe String)
        bhv $ toHtml $ length_b name
```

The more elaborate possibility is sending some data through the POST request and then possibly processing the answer, once received; this is accomplished using the function

$$post :: (BJSON\ \alpha) \Rightarrow String \rightarrow Event\ [(String, String)] \rightarrow HtmlM\ (Bhv\ (Maybe\ \alpha))$$

The first parameter is again the name of the request that will be used for the `q` URL parameter. The second one is an event providing the data that would be actually sent; the format (list of pairs) is the same as what we get in the event generated by a form element, so that one may be used directly; other means of creating such event may by of course used as well. The request is sent once for each occurrence of that event.

Note that the resulting type is encapsulated in the *HtmlM* monad; this forces the programmer to embed this call into the created HTML tree—it does not generate any additional HTML code, but does ensure that the requests are sent as they should be regardless of the use and evaluation of the results. The result itself is a behaviour of type *Maybe*, which is *Nothing* initially, then *Just* with the last received response.

Suppose that for the request `?q=sum`, the server sends the sum of two values, `x` and `y`, sent in the POST body. Then a summation using the server for the computation can be implemented thus:

```
span_ $ do
        request ← form $ do
                  textfield ! name "x"
                  textfield ! name "y"
        result ← post "sum" request :: HtmlM (Bhv (Maybe Int))
        bhv $ toHtml result
```

3.3 Value recursion in HtmlM

In certain situations, it may be needed to work with the behaviours and events generated by various HTML elements in the contents of those elements themselves (like using the data sent from a form to alter some information in the form itself) or to use such values in mutually recursive fashion. This is not possible in the standard Haskell[1], where the scope of variables defined in the *do* notation is strictly from that point downwards, but can be achieved using *value recursion* [6] and *do rec* notation [15].

The first thing we need to do for this system to work is to define sort of fixpoint operator, called $mfix :: (MonadFix\ m) \Rightarrow (\alpha \rightarrow m\ \alpha) \rightarrow m\ \alpha$, which should again in a sense return a least defined fixed point of its argument and is subject to certain laws [6]. Since the *HtmlM* monad is basically a slightly modified state monad, the definition of *mfix* for *State* translates simply for *HtmlM* as well, we just need to be sometimes careful in definitions of certain internal functions not to introduce unnecessary dependencies upon the state, so the recursive definitions work in as many situations as they can.

[1] At least not in sufficiently elegant manner.

With these, if we have some function *check* that checks the form data for some errors and returns *True* or *False* indicating the state, we can write:

```
check :: Bhv [(String, String)] → Bhv Bool
check = ...
div_ $ do
  rec request ← form do
      ...    — form contents
      ite_b (timed_b true_b id $ λ_ → fmap_b check request) ...
           — show some information on error
```

The $ite_b :: Bhv\ Bool \to Bhv\ \alpha \to Bhv\ \alpha \to Bhv\ \alpha$ is classical if-then-else construct, distinct from $bool_b$ only in the order of parameters; $fmap_b$ is behaviour version of *fmap*, here used to map over the *Timed* that is present in the request from the form (and $timed_b$ is the destructor for that type).

If we introduce one more function

```
until :: Bhv (HtmlM α) → Bhv (Maybe Html) → Bhv (HtmlM α)
```

which behaves as its first parameter when the second one is *Nothing* and switches to the *Just* of it whet it is available—it is almost like $\lambda x \to maybe_b\ x\ id$—but always keeps the inner value of the first parameter and as such can not be implemented using the functions provided so far, then we can create a form that is replaced once sent:

```
div_ $ do
  rec request ← bhv $ (cb $ form ...)
                      `until`
                      (timed_b nothing_b (λ_ _ → just "sent") request)
  ...
```

Complete examples will be given in the chapter 5 and provided along with the source code of the library.

4. Implementation

Here will be described the implementation aspects of the concepts presented in this work. For all the details, see the source code and the comments there, because we shall not reproduce it here all.

4.1 Behaviours in JavaScript

We will start with the description of the run-time JavaScript representation of ordinary values and behaviours. Regardless of concrete presentation of individual values, all are wrapped in *thunks* used to implement lazy evaluation. Each thunk is either an unevaluated expression or an already computed value; specifically, they are constructed with a function computing their value and equipped with a method *get* that returns the computed value directly if available, or calls the assigned function first.

Inside these thunks we have the actual data; for some, we use the native JavaScript representation: *true* and *false* for the values of type *Bool*, numbers for *Int*, strings for *JSString* or objects for *JSObject*. To encode the various algebraic data types, we use JavaScript objects, where the name of the constructor is used as a key for an array holding the parameters.

We use the *jQuery framework* [19] to abstract from differences among various web browsers regarding the handling of DOM or other aspects, and also to actually represent the HTML snippets (the values of type *Html*) in JavaScript and to manipulate those.

Among the points of starting the definitions with behaviour functions was that now, those are the only primitives we have to implement here (and do not need to be concerned with behaviours, functions of behaviour with various numbers of parameters and so on).

The behaviour functions are functions that take a parameter of type α, return a value of type β and also depend on time. They will be represented as objects with a method *compute*, which takes only one parameter—the value of type α—while the time is passed implicitly, in a global variable, since all the computations are performed in the current time. The time itself in just a discrete (integer) value, incremented whenever something affecting a behaviour (user input, response from a server) happens.

Behaviours are meant to be changing in time, so we need a system that makes it possible. For this purpose we establish a dependency relation among behaviours—each behaviour function has a list of its dependencies (for example the one constructed as a composition depends on the two parts) forming a graph (which may, due to the value recursion, even contain cycles). When happens something, which changes any of the behaviours, such one is marked as invalid and so are all those depending on it (using reverse dependencies, which are also stored). Whenever a behaviour that represents an HTML snippet or a POST request is invalidated, that one is recomputed and appropriate action—replacing the HTML or sending the request—is carried out.

Individual behaviour functions are created in two steps: first the object is created and common initializations is performed, after that (when all relevant objects are created, so even mutual dependencies may be established) is done the initialization specific for each type of behaviour function. For the purpose of the second phase, we have a library of all the primitives used; implementation of them is generally straightforward, so it does not need to be analysed here.

4.2 The HtmlM monad

Before defining the *HtmlM* monad itself, we first need to show how the HTML structure is described:

```
data Attribute = AttrVal String String
               | AttrBool String


data HtmlStructure = Tag String [Attribute] [HtmlStructure]
                   | TagVoid String [Attribute]
                   | Doctype
                   | Text String
                   | Behaviour Int
```

The meaning of the constructors of *Attribute* (name-value pair and boolean attribute) and the first four of *HtmlStructure* (tag with content, without content, doctype and text node) is clear and would not be commented further. *Behaviour i* marks a position where some HTML behaviour (identified by its ID in the parameter i) is placed; when the textual representation of HTML page is created, some placeholder is included and then replaced by the actual dynamic content.

The *HtmlM* monad is then defined as:

```
data HtmlM a =
  HtmlM (HtmlState → (a, ([HtmlStructure], HtmlState)))
```

The *HtmlState* is just a record type, which holds several information internal for the working of our library. Those include a counter for generating unique (per rendering) numbers, the list of behaviour functions used in given snipped and also the assignment of HTML-generating behaviours. The instances are:

```
instance Functor HtmlM where
  fmap f (HtmlM hf) = HtmlM $ λs → (λ(x, hs) → (f x, hs)) (hf s)

instance Monad HtmlM where
  return x = HtmlM $ λs → (x, ([], s))
  (HtmlM f) >>= g = HtmlM $ λs → let
                                   (x, (cs, s')) = f s
                                   (HtmlM g') = g x
                                   (y, (ds, s'')) = g' s'
                                 in
                                 (y, (cs ++ ds, s''))
```

Since the empty list is the neutral element for the function $+\!\!+$ and *return* does not change the internal state, the left and right identity monad laws are satisfied; the associativity of $+\!\!+$ and function composition gives the third one. Note also that the resulting $\gg$ concatenates the [*HtmlStructure*] inside two values of type *Html* as we required earlier.

In order to turn the values of type *Html* into actual HTML code, we provide the function

$$render :: Html \rightarrow IO\ String$$

The *IO* type is used, because the string representation of a page may expose differences between semantically equal values, as we pointed out earlier in section 1.2, and we do not want to break referential transparency.

During rendering of a page, the initialization code for the used behaviours is also generated. The list of them is kept in the internal state of the *HtmlM* monad, so *render* evaluates the whole thing, giving it an empty state at the beginning, and along with generating the HTML code goes through the list of behaviours and outputs also the JavaScript code for them. For each behaviour, it keeps an ID, with which it is referenced from other places, the name of the JavaScript initialization function with necessary parameters and also information of possible dynamic parts of the page for which it may be responsible.

4.3 Behaviours in Haskell

In Haskell, we represent the behaviour functions as expressions of the type

$$\begin{aligned}\textbf{data}\ BhvFun\ \alpha\ \beta = {} & Prim\ (\alpha \rightarrow \beta)\ (HtmlM\ (String,\ [RawJS])) \\ \mid {} & Assigned\ (\alpha \rightarrow \beta)\ (Int,\ Int) \\ \mid {} & (\alpha \sim \beta) \Rightarrow BhvID\end{aligned}$$

The first parameters of the *Prim* and *Assigned* constructors are just Haskell functions to which it evaluates when the *unsafeBfEval* is applied. The second parameter of *Prim* is name of a JavaScript initialization function and a list of its parameters (*RawJS* is just a newtype wrapper around *String*), which can be got once the expression is evaluated inside the HtmlM monad. The numbers in the constructor *Assigned* identifies some behaviour function already registered in a particular instance of *HtmlM*; since there is no function that would get values out of that monad, those will stay there. *BhvID* is just an identity function.

We can also get the JavaScript representation of a behaviour function (i.e. the name of the variable containing the object), but that is possible only by evaluating the whole thing in the *HtmlM* monad, since *Assigned* values are not valid elsewhere. This is the reason, why the method of the type class *BhvValue* mentioned in the section 2.1 is defined as:

$$\begin{aligned}&\textbf{class}\ BhvValue\ \alpha\ \textbf{where} \\ &\quad bhvValue :: \alpha \rightarrow HtmlM\ RawJS\end{aligned}$$

Actual values (behaviour functions) are created either using various primitives (as the name of the *Prim* constructor suggests) or as special behaviours in the

HtmlM monad, for which the *Assigned* constructor is intended. The primitives have to be accompanied by a JavaScript function, which is determined by the first part of the second argument of *Prim*. For example, the definition of a composition of two behaviour functions may look basically like:

```
g `compose` f = Prim (unsafeBfEval g . unsafeBfEval f) $ do
  jf ← bhvValue f
  jg ← bhvValue g
  return ("compose", [jf, jg])
```

On the other hand, the the behaviours like the one generated from a *textfield* are created inside the *HtmlM* monad, where they are registered into the state (and paired with a HTML element from which they are generated) and then returned as values constructed with *Assigned*.

4.3.1 Lifting functions

In section 1.4 we used the fact that functions between behaviours can be lifted to a behaviour, that is that there exists a function

$$cb :: \forall\alpha\,\beta.\,(Bhv\,\alpha \to Bhv\,\beta) \to Bhv\,(Bhv\,\alpha \to Bhv\,\beta)$$

This is achieved by the fact that in the end, behaviours are evaluated inside the HtmlM monad (what we need is the database of assigned behaviours). When the code for the behaviour $cb\,f$ is created, it is placed inside an anonymous JavaScript function taking one parameter (representing the value of type $Bhv\,\alpha$); a behaviour representing this parameter is inserted into the list of behaviours and assigned a number and then passed to the function f, yielding a value of type $Bhv\,\beta$, which can be finally converted to JavaScript (inside *HtmlM*) and returned from the function (in JavaScript).

For the purpose of accessing both behaviours assigned inside such functions and outside of it, we keep a "recursion counter" in the state of *HtmlM*, which is incremented for each such nested call. Individual behaviours are then identified by two numbers—recursion level and per-function unique id—which is the reason for the pair of *Int*s in the constructor *Assigned*.

We can easily generalize this process to work with functions of arbitrary arity. Moreover, very similar procedure allows us to define a function

$$\begin{aligned} cbf :: (Bhv\,\alpha_1 \to Bhv\,\alpha_2 \to \cdots \to Bhv\,\alpha_n) \to \\ Bhv\,(\alpha_1 \to \alpha_2 \to \cdots \to \alpha_n) \end{aligned}$$

This is used to avoid some unnecessary wrapping and unwrapping of behaviours during a manipulation with such functions.

5. Examples

We first summarize several examples already encountered into a fully working code:

```
page :: Html
page = html $ do
  head_ $ do
    title "Page title"
  body $ p $ do
      — displaying the value and the length of a textfield:
    t ← textfield
    bhv $ toHtml t
    bhv $ toHtml $ length_b t
    br
      — showing a value sent by the server:
    let value = sget "value" :: Bhv (Maybe Int)
    bhv $ toHtml value
    br
      — value incremented for each click on the button:
    b ← button ! value "+1"
    bhv $ toHtml $ evfold (const (+)) (0 :: Bhv Int) $ fmap_b (const 1) b
```

Such code can then be accompanied by a *main* function:

```
main :: IO ()
main = putStr =≪ render page
```

in which case it can be compiled and executed to generate the code for the page, or it can be made a part of some server written in Haskell and served from that.

In the next example, we will show how to navigate over several pages using anchor (`<a>`) tags. In our library, these come in two flavours: first is ordinary tag generated by the *a* function, which can be used for external references if the *href* attribute is set; the second one is *ae* (for *anchor event*) and that, instead of redirecting the browser, generates events (with values from a behaviour given as parameter) whenever it is clicked on. The type signature is:

```
ae :: Bhv String → Html → HtmlM (Event String)
```

The behaviour in the parameter is intended to make it easier to aggregate several such links using *evmerge*, where the resulting event holds the identification of the last-clicked element.

Suppose now, that we have several pages already written:

```
page1, page2, page3 :: Html
page1 = div_ $ "Page 1"
page2 = div_ $ "Page 2"
page3 = div_ $ "Page 3"
```

We aggregate them with identifiers (here, it is show with a fixed number of elements, but it can as well work for dynamically generated list) and make them switchable using links:

```
pages :: Bhv [(String, Html)]
pages = cb $
  [ ("first", page1)
  , ("second", page2)
  , ("third", page3)
  ]
mainPage :: Html
mainPage = html $ do
  head_ $ do
    title "Page title"
  body $ do
    a1 ← ae "first" $ "First page"
    a2 ← ae "second" $ "Second page"
    a3 ← ae "third" $ "Third page"
    br
    bhv $ timed_b ""(λ_ k → maybe_b "" id (lookup_b k pages)) $
      evmerge const a1 (evmerge const a2 a3)
```

The last example shows a registration form where a username and password is given. Before sending the data, the form checks the password if it is the same in the both provided fields and alerts a user if not; once sent, it replaces itself with a text saying so and finally changes when a response is received:

```
page :: Html
page = html $ do
  head_ $ do
    title "Page title"
  body $ do
    rec result ← post "register" formData' :: HtmlM (Bhv (Maybe Int))
        (formData, formData') ← fmap (id &&& eguard check) $ bhv $ (
            cb $ form $ do
              str "Name:"
              textfield ! name "name"
              br; str "Password:"
              textfield ! name "pass" ! type_ "password"
              br; str "Check:"
              textfield ! name "pass-check" ! type_ "password"
              br; bhv $ bstr "Passwords differ" `displayUnless`
                timed_b true_b (const check) formData
              br; submit
            ) `until` (
              fmap_b (const "Sending...") $ e2m formData'
            ) `until` (
              fmap_b toHtml result
            )
```

```
  — Converts an event to a behaviour of Maybe
e2m :: Event α → Bhv (Maybe α)
e2m = timed_b nothing_b (const just_b)

  — Displays content only, if the condition does not hold
displayUnless :: (ToHtmlBhv α) ⇒ Bhv α → Bhv Bool → Bhv Html
displayUnless what = toHtml ∘ bool_b nothing_b (just_b what)

  — Guards an event—eliminates occurrences not satisfying given condition
eguard :: (Bhv α → Bhv Bool) → Event α → Event α
eguard f tx = ite_b (timed_b false_b (const f) tx) tx notYet_b

  — Check, if the passwords match
check :: Bhv [(String, String)] → Bhv Bool
check x = lookup "pass" x == lookup "pass-check" x
```

Conclusion

We described and implemented a library, which can be used to write dynamic web applications and is based on the principles of functional reactive programming, providing most of the combinators available in other FRP implementations. Although it is necessary to use functions written specifically for the purpose of working within this framework, it does not, in the end, limit the expressive power of the system, since potentially any algebraic data types can be manipulated and the general recursion is also available.

In doing so, we needed to develop a different approach to the reactive system, because the traditional monadic or arrowized ways did not work within the given constraints. Using other concepts from category theory, however, enabled us to describe the properties of our system in a fine enough detail to be useful.

Entirely different way of solving the problem could be using a special Haskell-to-JavaScript compiler. In such a system, one would write separately the code for the client and for the server, the client code would be then compiled directly into a JavaScript code and later served to the user as such.

Our approach, however, gives more possibilities: apart from being once compiled and then always sent in that form, expressions in our framework may be evaluated by the client or the server depending on situation; for example if some informations are know before sending the page, they could be included directly with the first response. By evaluating as much as can be, it may be possible to create an application working even in browsers without JavaScript, while retaining the dynamic elements in those that support it.

However, the library provided with this work is rather a proof-of-concept, and also it can be used to create simple dynamic web applications, for it to be suitable for larger projects, it would probably need improved interface and better optimizations. Also, the possibilities mention in the previous paragraph are not currently provided; those will be subject of further development.

Bibliography

[1] Adámek, J., Herrlich, H., Strecker, G. *Abstract and Concrete Categories* [online]. on-line edition, January 12, 2004 [cit. 2011-08-01]. ⟨http://katmat.math.uni-bremen.de/acc/acc.pdf⟩.

[2] Amsden, Edward. *A Survey of Functional Reactive Programming: Concepts, Implementations, Optimizations, and Applications*. May 20, 2011 [cit. 2011-08-01].
⟨http://www.cs.rit.edu/~eca7215/frp-independent-study/Survey.pdf⟩.

[3] Barendregt, H. P. *The lambda calculus: its syntax and semantics*. Paperback edition. Amsterdam: North-Holland, 1984. ISBN 0-444-87508-5

[4] Elliott, C., Hudak, P. Functional Reactive Animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*. New York: ACM, 1997. pp263–273.

[5] Elliott, Conal. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*. New York: ACM, 2009. pp25–36.

[6] Erkök, Levent. *Value Recursion in Monadic Computations*. Portland, 2002. A dissertation at the OGI School of Science and Engineering at Oregon Health and Science University. Advised by Dr. John Launchbury.

[7] Hudak, P., Courtney, A., Nilsson, H. and Peterson, J.: Arrows, Robots, and Functional Reactive Programming. In *Advanced Functional Programming, 4th International School, volume 2638 of LNCS*. Berlin: Springer-Verlag, 2003. pp159–187.

[8] Hughes, John. Generalising Monads to Arrows. In *Science of Computer Programming, volume 27*. May 2000. pp67–111.

[9] Kmett, Edward A. *Categories* [Haskell package]. Ver. 0.58.0. Jul 11, 2011 [cit. 2011-08-01].
Available at ⟨http://hackage.haskell.org/package/categories-0.58.0⟩.

[10] Lambek, J., Scott, P. J. *Introduction to higher order categorical logic*. Paperback edition, 3rd printing, 1994. Newcastle upon Tyne: Athenæum Press Ltd. ISBN 0-521-35653-9.

[11] Nilsson, H., Courtney, A., Peterson, J. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*. New York: ACM Press, 2002. pp51–64.

[12] Paterson, Ross. A new notation for arrows. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*. New York: ACM, 2001. pp229-240.

[13] Sheard, Tim, Peyton Jones, Simon. Template metaprogramming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*. New York: ACM Press, 2002. pp1–16.

[14] Van der Jeugt, Jasper, Meier, Simon. *BlazeHtml* [Haskell package]. Ver. 0.4.1.1. Apr 8, 2011 [cit. 2011-08-01].
Available at ⟨http://hackage.haskell.org/package/blaze-html-0.4.1.1⟩.

[15] *GHC syntactic extensions.* [online]. [cit. 2011-08-01].
⟨http://www.haskell.org/ghc/docs/7.0.4/html/users_guide/syntax-extns.html⟩.

[16] *Happstack* [Haskell library]. Ver. 6.0.3, Jul 27, 2011 [cit. 2011-08-01].
Available from ⟨https://happstack.com⟩

[17] *Introducing JSON* [online]. [cit. 2011-08-01]. ⟨http://json.org/⟩.

[18] *Json* [Haskell package]. Galois Inc.: May 12, 2010 [cit. 2011-08-01].
Available at ⟨http://hackage.haskell.org/package/json⟩.

[19] *jQuery* [JavaScript library]. Ver. 1.6.2, ©2011 [cit. 2011-08-01].
Available from ⟨http://jquery.com/⟩