# Sunroof: A Monadic DSL to Generate JavaScript

Jan Bracker[1,2] and Andy Gill[1]

[1] ITTC / EECS
The University of Kansas
Lawrence, KS 66045

[2] Institut für Informatik
Christian-Albrechts-Universität
Kiel, Germany

**Abstract.** Sunroof is a Haskell-hosted Domain Specific Language (DSL) for generating JavaScript. Sunroof is build on top of the JavaScript monad, which, like the Haskell `IO`-monad, allows access to external resources, but specifically JavaScript resources. As such, Sunroof is primarily a feature-rich foreign-function API to the browser's JavaScript engine, and all the browser-specific functionality, including HTML-based rendering, event handling, and drawing to the HTML5 canvas.
In this paper, we give the design and implementation of Sunroof. Using monadic reification, we generate JavaScript from the a deep embedding of the JavaScript monad. The Sunroof DSL has the feel of native Haskell, with a simple Haskell-based type schema to guide the Sunroof programmer. Furthermore, because we are generating code, we can offer Haskell-style concurrency patterns, such as MVars and Channels. In combination with a web-services package such as Scotty, the Sunroof compiler offers a robust platform to build interactive web applications, giving the ability to interleave Haskell and JavaScript computations with each other as needed.

**Keywords:** DSLs, JavaScript, Web Technologies, Cloud Computing

## 1 Introduction

JavaScript is an imperative language with access to a wide range of established and useful services, e.g. graphical canvases and event handling. It also provides features that are associated with functional languages, such as first-class functions. We want to express JavaScript in Haskell, adding use of Haskell's static typing, and gaining access to JavaScript services in the browser from Haskell.

Sunroof was developed to tackle this goal. The small example in Fig. 1 shows how Sunroof feels. The expected JavaScript output is shown on the right. But what makes this useful? We produced code that is shorter than the Sunroof code on the left and can easily be written by hand.

Sunroof's approach has several advantages. It introduces a threading model and abstraction similar to Haskell's. This allows programmers to use common

```
jsCode :: JS t ()
jsCode = do
  name <- prompt "Your name?"          var v0 = prompt("Your name?");
  alert ("Your name: " <> name)        alert("Your name: " + v0);
```

Fig. 1: Sunroof program and the expected JavaScript on the left.

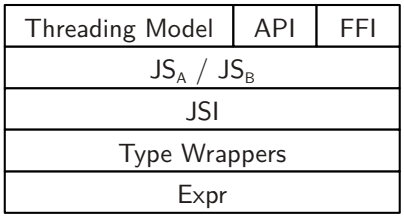


Fig. 2: The structure of Sunroof.

Haskell techniques when writing JavaScript. We also utilize the static type system to support us when writing code, e.g. the functions in Fig. 1 have the following signatures:

```
prompt :: JSString -> JSString -> JS t JSObject
alert  :: JSString -> JS t ()
```

This allows the type checker to detect a wide class of malformed JavaScript programs. Using a deep embedding gives opportunity for optimizations when producing the actual JavaScript. At the same time, Sunroof offers an interface that is close to actual JavaScript making it easy to use. The interface is easily extendable through a foreign-function interface. We use a monad [22] to reflect the imperative nature of JavaScript.

Fig. 2 shows Sunroof's structure. We will cover each of the layers throughout the paper:

- The semantics and implementation of the `JS`-monad is explained in Section 2, along with a description of how we solved the problem of constraining types involved in the monadic computations.
- Section 3 will discuss how we annotate JavaScript objects with types using wrappers and offer the possibility to add custom types later on.
- The special role of functions, continuations and how we model them as first-class values in Haskell and JavaScript will be covered in Section 4.
- The two threading models offered by Sunroof are explained in Section 5.
- Section 6 introduces Sunroof's foreign-function interface.
- Translation of Sunroof to JavaScript is handled in Section 7. We will explain the compilation of selected language constructs. This is especially interesting in the light of our use of continuations and their translation to JavaScript.
- The ability to interleave Haskell and JavaScript computations as needed through the Sunroof server will be highlighted in Section 8.
- Section 9 will cover a small application written in Sunroof to survey how usable Sunroof is in the context of application development.

```
data JSI :: T -> * -> * where
  JS_Invoke   :: (SunroofArgument a, Sunroof r)
              => a -> JSFunction a r -> JSI t r
  JS_Function :: (SunroofArgument a, Sunroof b)
              => (a -> JS A b) -> JSI t (JSFunction a b)
  JS_Branch   :: (SunroofThread t, SunroofArgument a, Sunroof bool)
              => bool -> JS t a -> JS t a  -> JSI t a
  JS_Assign_  :: (Sunroof a) => Id -> a -> JSI t ()
  ...
```

Fig. 3: Parts of the JavaScript instruction data type (`JSI`).

## 2 The JavaScript Monad

The imperative nature and side-effects of JavaScript are modeled through the `JS`-monad. It resembles the `IO`-monad, but has an extra phantom argument [18] to decide which threading model is used. For now we can ignore this extra argument. It will be discussed in Section 5.

The basic idea is that a binding in the `JS`-monad becomes an assignment to a fresh variable in JavaScript. This allows the results of previous computations are passed on to later ones. Fig. 1 gives an example. The binding `name` is translated to the freshly generated variable `v0`.

This simple example displays a challenging problem. Where does `v0` come from? The bind inside the monadic `do` is fully polymorphic over `a`.

```
(>>=) :: JS t a -> (a -> JS t b) -> JS t b
```

Thus we do not know how to create a value of type `a`. What we want is:

```
(>>=) :: (Sunroof a) => JS t a -> (a -> JS t b) -> JS t b
```

where `Sunroof` constrains the bind to arguments for which we can generate a variable. It turns out that a specific form of normalization allows the type `a` to be constrained and `JS` to be an instance of the standard monad class [26]. Through this keyhole of *monadic reification*, the entire Sunroof language is realized.

The normalization is done through Operational [3, 2]. It provides the `Program`-monad that can be equipped with custom primitives. We represent these primitives with the JavaScript instruction (`JSI`) type shown in Fig. 3. It represents the abstract instructions that are sequenced inside the `Program`-monad. The parameter `t` in `JSI t a` again represents the threading model and can be ignored up to Section 5. The type `a` represents the primitive's return value. `JS_Invoke` calls a function that has been created with `JS_Function`. Branches are represented with `JS_Branch`. Assignments to a variable are represented by `JS_Assign_`.

Without the alternative threading model, the normalization through Operational would be enough to offer a monad suitable for Sunroof. Due to our threading plans, there is more than just normalization going on behind the scenes. The `JS`-monad is a continuation monad over the `Program`-monad.

```
data JS :: T -> * -> * where
  JS :: ((a -> Program (JSI t) ()) -> Program (JSI t) ()) -> JS t a
  ...
```

The monad instance used is the standard implementation of a continuation monad.

## 3 JavaScript Object Model

One goal of Sunroof is to use Haskell's type system to increase the correctness of expressed JavaScript. At the same time we cannot characterize all different types of objects in JavaScript, since users can create their own objects. Thus our system to type JavaScript needs to be extensible.

Our solution is to provide a basic `Expr`ession language to construct JavaScript expressions that have no associated type information. Simplified slightly, we have:

```
data Expr
  = Lit String        -- Precompiled (atomic) JavaScript literal
  | Var Id            -- Variable
  | Apply Expr [Expr] -- Function application
  ...
```

In reality, we abstract `Expr` over the recursive type, to facilitate the usage of observable sharing [15] and allow sub-expression computations to be shared.

This core expression type is then wrapped to represent a more specific type. Each of these wrappers implements the `Sunroof` type class.

```
class SunroofArgument a => Sunroof a where
  box   :: Expr -> a
  unbox :: a -> Expr
  ...
```

It marks these types as possible values in JavaScript. The `SunroofArgument` prerequisite permits them to be function arguments (Section 4).

An example of this is `JSString`, the representation of JavaScript strings.

```
data JSString = JSString Expr
instance Sunroof JSString where
  box                = JSString
  unbox (JSString e) = e
```

But what do we gain through a wrapper? We can provide specific functionality for each distinct type. Our example type `JSString` has a `Monoid` and an `IsString` instance that are not provided for other wrappers, e.g. `JSBool` or `JSNumber`. This approach was first introduced by Svenningsson [29].

Table 1 gives a summary of the prominent Sunroof types. Some types involve phantom types to give more type safety [7]. The smooth embedding of booleans

| Constraint | Sunroof Type $\tau$ | Haskell Analog $\tau_\uparrow$ | `js` |
|---|---|---|---|
| | `()` | `()` | ✓ |
| | `JSBool` | `Bool` | ✓ |
| | `JSNumber` | `Double` | ✓ |
| | `JSString` | `String` | ✓ |
| `Sunroof` $\alpha$ | `JSArray` $\alpha$ | `[`$\alpha_\uparrow$`]` | |
| `SunroofKey` $\alpha$ | `JSMap` $\alpha$ $\beta$ | `Map` $\alpha_\uparrow$ $\beta_\uparrow$ | |
| `Sunroof` $\beta$ | | | |
| `SunroofArgument` $\alpha$ | `JSFunction` $\alpha$ $\beta$ | $\alpha_\uparrow \rightarrow$ `JS`$_{\texttt{A}}$ $\beta_\uparrow$ | |
| `Sunroof` $\beta$ | | | |
| `SunroofArgument` $\alpha$ | `JSMVar` $\alpha$ | `MVar` $\alpha_\uparrow$ | |
| `SunroofArgument` $\alpha$ | `JSChan` $\alpha$ | `Chan` $\alpha_\uparrow$ | |

Table 1: Sunroof types and their Haskell pendant.

and numbers is done through the Boolean package [12].

Table 1 shows that most basic Haskell types have counterparts in Sunroof. To convert Haskell values into their counterparts, we provide the `SunroofValue` class.

```
class SunroofValue a where
  type ValueOf a :: *
  js :: (Sunroof (ValueOf a)) => a -> ValueOf a
```

The type function `ValueOf` [6] provides the corresponding Sunroof type. `js` converts a value from Haskell to Sunroof. By design `SunroofValue` only provides instances for values that can be converted in a pure manner. Some types in JavaScript are referentially transparent according to `==`, while others, like general objects, are not. As an example, if you call `new Object()`, twice you get two equivalent empty objects, but when compared by `==` they are different. They are not identical, because in this case reference equality is checked instead of value equality. We call this observable allocation and handle it as a side-effect which may only occur in the `JS`-monad.

This approach ensures to bind a new value to a variable, instead of creating copies of that value everywhere it is used. This resolves some unwanted macro behavior of Sunroof.

## 4 Functions and Continuations

Functions are first class values in Haskell and JavaScript. Sunroof represents function with the type `JSFunction` $\alpha$ $\beta$, which resembles a function $\alpha \rightarrow \beta$ in JavaScript. Since partial application is questionable in the context of JavaScript, we only permit uncurried functions. Thus a `Sunroof` $\alpha$ constraint would prohibit functions to take more than one argument. We introduced `SunroofArgument` to constrain the types that may be used as arguments for functions.

```
class SunroofArgument args where
  jsArgs   :: args -> [Expr]
  jsValue  :: (UniqM m) => m args
  ...
```

It converts each argument into its expression through `jsArgs` to supply the arguments to a function call. `jsValue` generates unique names for each argument, which is needed when compiling the function itself to a value.

Now we can provide more then one argument by providing `SunroofArgument` instances for tuples of Sunroof values.

```
instance (Sunroof a, Sunroof b) => SunroofArgument (a,b) where
  jsArgs (a,b) = [unbox a, unbox b]
  jsValue = liftM2 (,) jsVar jsVar
```

Remember that each `Sunroof` value already has to be a `SunroofArgument`, which enables us to pass a single argument to a function.

We can create functions with the `function` combinator.

```
function :: (SunroofArgument a, Sunroof b)
         => (a -> JS A b)  -> JS t (JSFunction a b)
```

As a function can have side-effects, its computation and result have to be expressed in the `JS`-monad. The creation of a function is considered a side-effect, due to observable allocation.

Function application is done through the `apply` or `$$` combinator, they are synonyms. Functions can only be applied in the `JS`-monad, since they can have side-effects.

```
apply, ($$) :: (SunroofArgument a, Sunroof b)
            => JSFunction args ret -> args -> JS t ret
```

Creation and application are implemented using the `JS_Function` and `JS_Invoke` instructions introduced in Fig. 3.

`JSContinuation` $\alpha$ is used to model continuations in the `JS`-monad. It was introduced to work with continuations in Sunroof. Technically they are only specializations of functions, but restricted to the second threading model. Continuations are meant to be a representation of side-effects – ongoing computations inside the `JS`-monad – and may not terminate, so they do not return a value. As with functions, there is a combinator to create and apply a continuation.

```
continuation :: (SunroofArgument a)
             => (a -> JS B ()) -> JS t (JSContinuation a)
goto         :: (SunroofArgument a)
             => JSContinuation a -> a -> JS t b
```

The presented `goto` should not be considered harmful [10]. It calls a continuation, as `apply` calls functions. The difference is that a call to `goto` will never return,

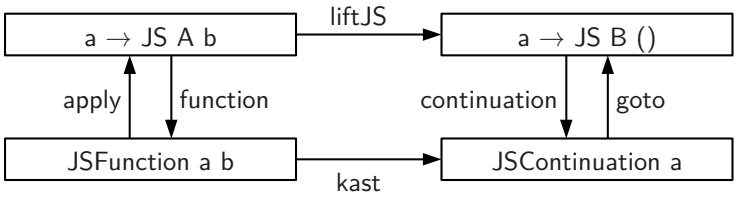


Fig. 4: How functions and continuations relate between the Haskell and Sunroof domain.

as it executes the given continuation and abandons the current one. This allows `goto` to be fully polymorphic on its result.

Access to the current continuations is given through the powerful call-with-current-continuation combinator `callcc`.

```
callcc :: SunroofArgument a
       => (JSContinuation a -> JS B a) -> JS B a
```

The current continuation models everything that would usually happen after the call to `callcc`.

```
callcc f = JS $ \ k -> unJS
    (continuation (\a -> JS $ \ _ -> k a) >>= f) k

unJS :: JS t a -> (a -> Program (JSI t) ()) -> Program (JSI t) ()
```

The implementation of `callcc` is interesting, because it shows how the `Program`-continuation is translated into a `JSContinuation` that is passed to the given function `f`. Section 5 will show why this function is important for Sunroof and what it is used for.

Functions and continuations are similar and connected to each other, as can be seen in Fig. 4. We can go back and forth between the Haskell and the Sunroof representation of a function or continuation. But once a function is specialized to a continuation, it is not possible to go back, because continuations only model the side-effect, but do not return anything.

## 5 Threading Models

JavaScript uses a callback centric model of computation. There is no concurrency, only a central loop that executes callbacks when events occur.

In contrast, Haskell has real concurrency and wide-spread abstractions for synchronization, e.g. `MVar`s and `Chan`s [17]. So the question arises: do we generate atomic JavaScript code, and keep the callback centric model, or generate JavaScript using CPS [8], and allow for blocking primitives and cooperative concurrency. The latter, though more powerful, precluded using the compiler to generate code that can be cleanly called from native JavaScript. Both choices had poor consequences.

We decided to explicitly support both, and make both first-class threading strategies in Sunroof. In terms of user-interface, we parameterize the `JS`-monad with a phantom type that represents the threading model used, with `A` for `A`tomic, and `B` for `B`locking threads. Atomic threads are classical JavaScript computations that cannot be interrupted and actively use the callback mechanism. Blocking threads can support suspending operations and cooperative concurrency abstractions as known from Haskell. By using phantom types, we can express the necessary restrictions on specific combinators, as well as provide combinators to allow both types of threads to cooperate.

The blocking model hides the callback mechanism behind abstractions. This implies that every atomic computation can be converted into a blocking computation. `liftJS` achieves this.

```
liftJS :: Sunroof a => JS A a -> JS t a
```

When suspending, we register our current continuation as a callback to resume later. This gives other threads (registered continuations) a chance to run. Of course, this model depends on cooperation between the threads, because a not terminating or suspending thread will keep others from running.

There are three main primitives for the blocking model:

```
forkJS      :: SunroofThread t1 => JS t1 () -> JS t2 ()
threadDelay :: JSNumber -> JS B ()
yield       :: JS B ()
```

They can all be seen as analogues of their `IO` counterparts. `forkJS` resembles `forkIO`. It registers the continuation of the given computation as a callback. `yield` suspends the current thread by registering the current continuation as a callback, giving other threads time to run. `threadDelay` is a form of `yield` that sets the callback to be called after a certain amount of time. We rely on the JavaScript function `window.setTimeout` [1] to register our callbacks.

The class `SunroofThread` offers functions to retrieve the current threading model (`evalStyle`) and to create a possible blocking computation (`blockableJS`).

```
class SunroofThread (t :: T) where
  evalStyle   :: ThreadProxy t -> T
  blockableJS :: (Sunroof a) => JS t a -> JS B a
```

Based on these primitive combinators, we also offer a Sunroof version of `MVar` and `Chan`: `JSMVar` and `JSChan`.

```
newMVar      :: (SunroofArgument a) => a -> JS t (JSMVar a)
newEmptyMVar :: (SunroofArgument a) => JS t (JSMVar a)
putMVar      :: (SunroofArgument a) => a -> JSMVar a -> JS B ()
takeMVar     :: (SunroofArgument a) => JSMVar a -> JS B a

newChan   :: (SunroofArgument a) => JS t (JSChan a)
writeChan :: (SunroofArgument a) => a -> JSChan a -> JS t ()
readChan  :: (SunroofArgument a) => JSChan a -> JS B a
```

Both implementations use arrays to store the waiting readers and writers in the form of continuations. Note that all functions are able to handle `SunroofArgument`s, not just `Sunroof` types. This is possible, because the computations themselves (their current continuation) are stored in the lists through `callcc`. When arguments are written, either the waiting continuation is called with those arguments or a new continuation that applies an incoming one with those arguments is created.

## 6 Foreign Function Interface

Sunroof also offers a foreign function interface, which enables us to easily access predefined JavaScript. There are four core functions:

```
fun    :: (SunroofArgument a, Sunroof r)
       => String -> JSFunction a r
object :: String -> JSObject
new    :: (SunroofArgument a)
       => String a -> JS t JSObject
invoke :: (SunroofArgument a, Sunroof o, Sunroof r)
       => String -> a -> o -> JS t r
```

`fun` is used to create Sunroof functions from their names in JavaScript. This can happen in two ways: either to call a function in line, or to create a real binding for that function. As an example, the `alert` function can be called in line through `fun "alert" $$ "text"`, or you can provide a binding in form of a Haskell function for it.

```
alert :: JSString -> JS t ()
alert s = fun "alert" $$ s
```

Objects can be bound through the `object` function, e.g. the `document` object is bound through `object "document"`. Constructors can be called using `new`. To create a new object you would call `new "Object" ()`.

We can call methods of objects through `invoke`. Again, this can be used in line and to create a real binding. An inline use of this to produce `document .getElementById("id")` would look like this:

```
object "document" # invoke "getElementById" "id"
```

where `#` is just a flipped function application. To provide a binding to the `getElementById` method, one can write:

```
getElementById :: JSString -> JSObject -> JS t JSObject
getElementById s = invoke "getElementById" s
```

Providing actual bindings ensures that everything is typed correctly and prevents the need to resolve ambiguities through large type annotations inside of code.

The current release of Sunroof already provides bindings for most of the core browser API, the HTML5 canvas element, and some of the JQuery API.

## 7 The Sunroof Compiler

How do we compile Sunroof? Through the `JS`-monad we produce a `Program (JSI t) ()` instance. We translate such a program into a list of statements (`Stmt`) by matching over the `JSI` constructors.

```
data Stmt = AssignStmt Rhs Expr       -- Assignment
          | ExprStmt Expr             -- Expression as statement
          | ReturnStmt Expr           -- Return statement
          | IfStmt Expr [Stmt] [Stmt] -- If-Then-Else statement
          ...
```

The constructors of `Stmt` directly represent the different statements you can write in JavaScript. Operationals [3, 2] `Program` type is abstract and has to be converted to a `ProgramView` instance to work with. It provides the `:>>=` and `Return` constructors to pattern match on and translate the instructions inside.

```
compile :: Program (JSI t) () -> CompM [Stmt]
compile p = eval (view p)
  where eval :: ProgramView (JSI t) () -> CompM [Stmt]
        ...
```

The `CompM` state monad provides the compiler options and fresh variables.

To get started, `JS_Assign_` is translated in the following manner. Recall from Fig. 3 that it takes a variable name and the value to assign as arguments.

```
eval (JS_Assign_ v a :>>= k) = do
  (stmts0,val) <- compileExpr (unbox a) -- Expr -> CompM ([Stmt], Expr)
  stmts1       <- compile (k ())
  return ( stmts0 ++ [AssignStmt (VarRhs v) val] ++ stmts1)
```

First, we compile the expression `a` to assign into a series of statements that compute it together with the result value `val`. Next, we compile what happens after the assignment. Since the assignment produces unit, we can pass that to `k`. In the end, we concatenate all statements with the assignment in between.

Now we will look into the translation of branches. Recall the `JSI` constructor for branches from Fig. 3:

```
JS_Branch :: (SunroofThread t, SunroofArgument a, Sunroof bool)
          => bool -> JS t a -> JS t a  -> JSI t a
```

Fig. 5 shows how to compile a `JS_Branch`. We generate the statements for the branching condition. Then we generate unique variables for each of the returned values in either branch and use `bindResults` to assign them to the new variables. The function `extractProgramJS` passes the result of a computation `m` into the function `f` and closes the continuation in `JS` with `return`. The result is a `Program` containing all instructions of `m >>= k`.

```
extractProgramJS :: (a -> JS t ()) -> JS t a -> Program (JSI t) ()
extractProgramJS k m = unJS (m >>= k) return
```

```
eval (JS_Branch b c1 c2 :>>= k) = do
  (src0, res0) <- compileExpr (unbox b)
  res :: a <- jsValue
  let bindResults :: a -> JS t ()
      bindResults res' =
        sequence_ [ single $ JS_Assign_ v (box $ e :: JSObject)
                  | (Var v, e) <- jsArgs res `zip` jsArgs res' ]
  src1 <- compile $ extractProgramJS bindResults c1
  src2 <- compile $ extractProgramJS bindResults c2
  rest <- compile (k res)
  return (src0 ++ [ IfStmt res0 src1 src2 ] ++ rest)
```

Fig. 5: Naive translation of branches in Sunroof.

After the compilation of both branches, we translate the rest and create the list of statements in a canonical fashion.

This works perfectly if we are in the atomic threading model, but can fail when inside the blocking model. In the blocking model either branch may involve continuations. Each continuation describes the rest of the computation up to the end of that branch. It also captures the assignments at the end of the branch. Assignments captured inside a continuation are not visible outside of it. That means the variables are not visible after the branch and the code inside of `rest` refers to them although they are not defined in that scope.

Therefore, within the blocking threading model, we have to handle branches differently:

```
eval (JS_Branch b c1 c2 :>>= k) =
  case evalStyle (ThreadProxy :: ThreadProxy t) of
    A -> compileBranch_A b c1 c2 k
    B -> compileBranch_B b c1 c2 k
```

The call to `compileBranch_A` executes our naive definition from Fig. 5.

```
compileBranch_B b c1 c2 k = do
  fn_e <- compileContinuation $
            \a -> blockableJS $ JS $ \k2 -> k a >>= k2
  fn <- newVar
  (src0, res0) <- compileExpr (unbox b)
  src1 <- compile $ extractProgramJS (apply (var fn)) c1
  src2 <- compile $ extractProgramJS (apply (var fn)) c2
  return ( [mkVarStmt fn fn_e] ++ src0 ++ [ IfStmt res0 src1 src2 ])
```

We can see that the rest of our computation is captured in the continuation `fn_e`. It takes the results of a branch as arguments. A new variable `fn` is used to share the continuation in both branches. The key difference is the parameter to `extractProgramJS`. Instead of creating bindings to variables, we apply the produced continuation to the returned values. That passes them to the ongoing

computation. Of course, we could compile every branch with the continuation variant, but this would unnecessarily obfuscate the produced code.

The compiler offers two functions to compile either threading model:

```
sunroofCompileJSA :: Sunroof a
                  => CompilerOpts -> String -> JS A a  -> IO String
sunroofCompileJSB :: CompilerOpts -> String -> JS B () -> IO String
```

Notice that for the blocking threading model only unit may be returned due to their continuation-based nature. The string argument provides the name of the variable to which to bind the computation result.

## 8 The Sunroof Server

The Sunroof server provides infrastructure to send arbitrary pieces of JavaScript to a calling website for execution. So, it is possible to interleave Haskell and JavaScript computations with each other as needed. The three major functions provided are `sunroofServer`, `syncJS` and `asyncJS`.

```
type SunroofApp = SunroofEngine -> IO ()
sunroofServer :: SunroofServerOptions -> SunroofApp -> IO ()

syncJS  :: SunroofResult a
        => SunroofEngine -> JS t a -> IO (ResultOf a)
asyncJS :: SunroofEngine -> JS t () -> IO ()
```

`sunroofServer` starts a server that will call the given callback function for each request. `syncJS` and `asyncJS` allow the server to run Sunroof code inside the requesting website. `asyncJS` executes it asynchronously without waiting for a return value. In contrast to that, `syncJS` waits until the execution is complete and then sends the result back to the server. It is converted into a Haskell value that can be processed further. Values that can be converted to a Haskell type after a synchronous call implement the `SunroofResult` class. It maps the Sunroof type to a corresponding Haskell type through a type function and provides a function to convert the data to that type.

## 9 Case Study - A Small Calculator

To see how Sunroof works in practice, we will look into the experience we gathered when writing a small calculator for arithmetic expressions (Fig. 6). We use Sunroof to display our interface and the results of our computation. Haskell will be used to parse the arithmetic expressions and calculate the result. The Sunroof server will be used to implement this JavaScript/Haskell hybrid.

The classical approach to develop an application like this would have been to write a server that provides a RESTful interface and replies through a JSON

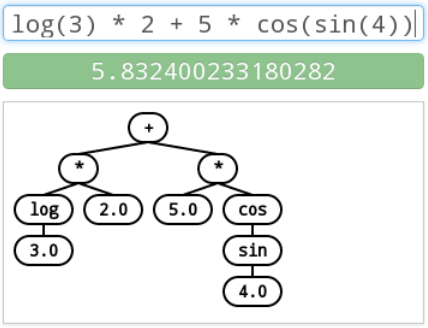


Fig. 6: The example application running on the Sunroof server.

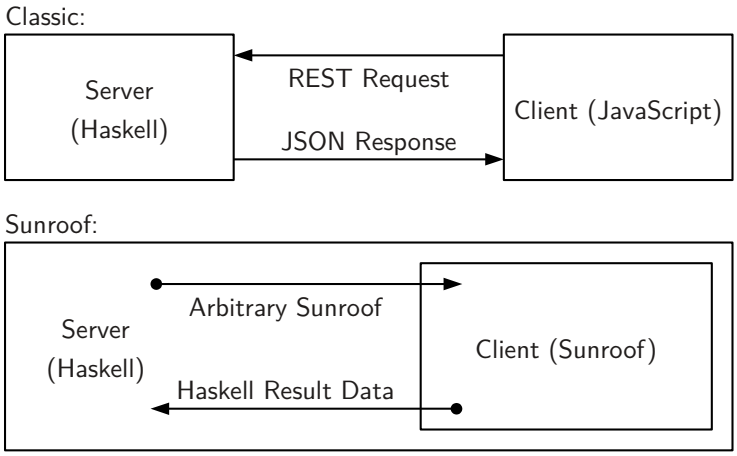


Fig. 7: Classical structure and Sunroof structure of a web application.

data structure. The client side of that application would have been written in JavaScript directly. This can be seen in Fig. 7.

How does Sunroof improve or change this classical structure? First of all, in Sunroof you write the client-side code together with your server application within Haskell. In our example, all code for the server and client is in Haskell. The control logic for the client side is provided through the server. This leads to a tight coupling between both sides. It forces both sides to work together correctly because they share types and interfaces. This also shows how Sunroof blurs the border between the server and client side. You are not restricted by an interface or language barrier. If you need the client to do something, you can just send arbitrary Sunroof code to execute in the client.

Following are a few statistics (Table 2) about how much code it took to write each part of the client.

The client-server response loop shuffles new input to the server and executes the response in the client.

Data conversion is needed, because pure Haskell data types cannot be handled in Sunroof and vice versa. There still exists a language barrier between

| Part of Application | Lines of Code | Percentage |
|---|---|---|
| Response loop | 25 | 6.5% |
| Data conversion | 85 | 22.0% |
| Rendering | 190 | 49.5% |
| Parsing and interpretation | 85 | 22.0% |

Table 2: Lines of code needed for the example.

JavaScript and Haskell. Code to convert between two essentially equal data structures on each side must be written, as well as representations of Haskell structures in Sunroof. However, there is great potential in automatically generating this code using techniques such as template Haskell [27].

The code for displaying the results is basically a transliteration of the JavaScript that you would write for this purpose. The transliteration used here is not very appealing. In the future, this code can be generated through higher-level libraries. Sunroof is intended to deliver a foundation for this purpose.

The rest of our code to parse the arithmetic expression and calculate result is classical Haskell code.

We have a lot of boilerplate code for data conversion that has potential to be generated. The transliteration of JavaScript into Sunroof has an overhead and tends to be verbose. This problem can be hidden through higher-level libraries using Sunroof as a backend. We can see that Sunroof has the potential to be a low-level interface to the browser's capabilities. It offers a solid and robust foundation to build more advanced systems that want to utilize the browser.

## 10 Conclusion

Sunroof took the key idea of monad reification and successfully created the `JS`-monad to describe computations in JavaScript. This work was mainly done by Farmer and Gill [13] and has been streamlined during the further development of Sunroof. By adding the concept of `JSFunction` and `JSContinuation`, there now is a connection between functions in the JavaScript and the Sunroof language space (Fig. 4). It is possible to go back and forth between both worlds. Combining both concepts, functions and the `JS`-monad, we were able to create a second implementation of the monad, this time based on the direct translation of continuations from Haskell to JavaScript. It enabled us to build a blocking threading model on top of JavaScript that resembles the model already known from Haskell. Based on this model and the provided abstraction over continuations, we can use primitives like `forkJS` or `yield`. Higher-level abstractions like `JSMVar` and `JSChan` are also available.

## 11 Related Work

There have been several attempts to translate Haskell to JavaScript. Prominent ones are the compiler backends for UHC [28] and GHCJS [24]. There are also projects like Fay [11] that compile subsets of Haskell to JavaScript or JMacro [5] which use quasiquotation [21] to embed a custom-tailored language into Haskell code.

At the same time there are also projects like CoffeeScript [4] or LiveScript [23] to build custom languages that are very similar to JavaScript but add convenient syntax and support for missing features.

Our approach to cooperative concurrency through continuations in JavaScript has has been used before [9, 25]. To our knowledge, creating a direct connection between Haskell and JavaScript continuations has not been attempted before.

Deep embeddings of monads based on data structures have been used before in Unimo [19] and Operational [3, 2]. The specific approach Sunroof takes by using GADTs has been discussed by Sculthorpe et al. [26] in detail.

The Sunroof server does not have the aim to provide a full-featured web framework, as HAppS, Snap or Yesod do. It only provides the infrastructure to communicate with the currently calling website through the Kansas comet [16] push mechanism [20]. Although all of the frameworks mentioned above would be able to implement this technique, to our knowledge, none of them has yet.

To our knowledge, Sunroof is the only library that supports generation of JavaScript inside of Haskell using pure Haskell in a type-safe manner. All other approaches discussed above either require a separate compilation step or introduce new syntax inside of Haskell.

There is an effort to generalize Active [30], a library for animations, and implement a backend based on Sunroof [14].

## 12 Acknowledgment

We want to thank Conal Elliott for his support in adapting the Boolean package [12] and helping us to extend it with support for deeply embedded numbers.

## References

1. HTML Living Standard - Timers, `http://www.whatwg.org/specs/web-apps/current-work/multipage/timers.html#timers`
2. Apfelmus, H.: (2010), `http://hackage.haskell.org/package/operational`
3. Apfelmus, H.: The Operational Monad Tutorial. The Monad.Reader 15, 37–55 (2010)
4. Ashkenas, J.: CoffeeScript, `http://coffeescript.org/`
5. Bazerman, G.: JMacro, `http://www.haskell.org/haskellwiki/Jmacro`
6. Chakravarty, M.M.T., Keller, G., Jones, S.P.: Associated Type Synonyms. SIGPLAN Not. 40(9), 241–253 (Sep 2005), `http://doi.acm.org/10.1145/1090189.1086397`

7. Cheney, J., Hinze, R.: First-Class Phantom Types (2003)
8. Claessen, K.: A Poor Man's Concurrency Monad. Journal of Functional Programming 9(03), 313–323 (1999)
9. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web Programming Without Tiers. In: Formal Methods for Components and Objects. pp. 266–296. Springer (2007)
10. Dijkstra, E.W.: Letters to the editor: go to statement considered harmful. Communications of the ACM 11(3), 147–148 (1968)
11. Done, C., Bergmark, A.: Fay, https://github.com/faylang/fay/wiki
12. Elliott, C.: Boolean, http://hackage.haskell.org/package/Boolean
13. Farmer, A., Gill, A.: Haskell DSLs for Interactive Web Services. In: Cross-model Language Design and Implementation (2012)
14. Gill, A.: sunroof-active, https://github.com/ku-fpg/sunroof-active
15. Gill, A.: Type-Safe Observable Sharing in Haskell. In: Proceedings of the Second ACM SIGPLAN Haskell Symposium. pp. 117–128. Haskell '09, ACM, New York, NY, USA (Sep 2009), http://doi.acm.org/10.1145/1596638.1596653
16. Gill, A., Farmer, A.: Kansas Comet, http://hackage.haskell.org/package/kansas-comet
17. Jones, S.P., Gordon, A., Finne, S.: Concurrent Haskell. In: Annual Symposium on Principles of Programming Languages: Proceedings of the 23 rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. vol. 21, pp. 295–308 (1996)
18. Leijen, D., Meijer, E.: Domain Specific Embedded Compilers. In: Domain-Specific Languages. pp. 109–122. ACM (1999)
19. Lin, C.: Programming Monads Operationally with Unimo. In: International Conference on Functional Programming. pp. 274–285. ACM (2006)
20. Mahemoff, M.: HTTP Streaming, http://ajaxpatterns.org/Comet
21. Mainland, G.: Why It's Nice to be Quoted: Quasiquoting for Haskell. In: Proceedings of the ACM SIGPLAN workshop on Haskell workshop. pp. 73–82. Haskell '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1291201.1291211
22. Moggi, E.: Notions of computation and monads. Information and Computation 93(1), 55–92 (1991)
23. Murakami, S., Ashkenas, J.: LiveScript, http://livescript.net/
24. Nazarov, V.: GHCJS Haskell to Javascript Compiler, https://github.com/ghcjs/ghcjs
25. Predescu, O.: Model-View-Controller in Cocoon using continuations-based control flow (2002), http://www.webweavertech.com/ovidiu/weblog/archives/000042.html
26. Sculthorpe, N., Bracker, J., Giorgidze, G., Gill, A.: The Constrained-Monad Problem (2013), submitted to the International Conference on Functional Programming
27. Sheard, T., Jones, S.P.: Template Meta-programming for Haskell. In: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell. pp. 1–16. Haskell '02, ACM, New York, NY, USA (2002), http://doi.acm.org/10.1145/581690.581691
28. Stutterheim, J.: Improving the UHC JavaScript backend. Tech. rep., Utrecht University (2012), http://www.norm2782.com/improving-uhc-js-report.pdf
29. Svenningsson, J., Axelsson, E.: Combining Deep and Shallow Embedding for EDSL. Presentation at Trends in Functional Programming, June (2012)
30. Yorgey, B.: Active, https://github.com/diagrams/active