Arrowized FRP Abstraction for Functional IoT Programs

Yahui Song and Wei-Ngan Chin

OVERVIEW
We propose a practical Arrowized Functional Reactive Programming\(\text{AFRP}\) abstraction and a prototype of the embedded domain-specific language (EDSL) in Haskell for Internet of Things (IoT) development which guides IoT developers to write high-order FRP programs directly.

- Immutability and static checking of purely functional programming are good for program reliability and maintainability.
- Continues time-varying values in FRP can be neatly mapped into IoT systems.
- Arrowized FRP further tackles the biggest drawback “space leak” problem of classical FRP.

Altogether, providing this abstraction not only simplifies the complex task of building responsive and type-safe IoT systems but also provides the ability to reason about IoT event streams.

REACTIVE PROGRAMMING & INTERNET OF THINGS
Assuming there is a dependency between temperature and the air conditioner (AC) such as if the temperature rose too high, the AC would be turned on automatically. From this functionality, we may abstract two modules, one is \texttt{Sensor}, one is \texttt{AC}, and the update method is needed to be defined.

<table>
<thead>
<tr>
<th>Passive Programming</th>
<th>Reactive Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Update method is defined in the \texttt{Sensor} module</td>
<td></td>
</tr>
<tr>
<td>• Remote setters and updates</td>
<td></td>
</tr>
<tr>
<td>• \texttt{Sensor} module is responsible for changes</td>
<td></td>
</tr>
<tr>
<td>• \texttt{AC} has no awareness on the dependency</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FRP &amp; Arrowized FRP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional Reactive Programming works with mutable values by recasting them as time-varying values, capturing the temporal aspect of mutability. FRP originally composes two particular abstractions: a continuous modelling of behaviors, and discrete reactive events from users or processes.</td>
</tr>
</tbody>
</table>

```
Behavior \(\alpha = \text{Time} \rightarrow \alpha\) - continuous 
Event \(\alpha = [\text{(Time, } \alpha)]\) - discrete 

Signal \(\alpha = \text{Time} \rightarrow \alpha\) 
SF \(\alpha \beta = \text{Signal } \alpha \rightarrow \text{Signal } \beta\)
```

Arrowized FRP rules out the “space leak” of functional programming by rewriting the signal functions as functions from one signal to another signal.

EXAMPLE – ENERGY AUTOMATION
Each temperature is a discrete Event while the state of the \texttt{AC} is a continuous Behavior.

```
data \texttt{Signal a = Signal \{func :: Time -> a\}}
type \texttt{Time = Double}
type \texttt{Temperature = Signal Float}
type \texttt{AC = Signal Bool}
```

SIGNAL STREAM GRAPHS
We start with reading from the world and end with writing to the world. Inside of this functional reactive IoT system, we only have pure functions and data streams pushing the side-effect to the edge of the system.

EDSL DESIGN
This embedded domain-specific language (EDSL) only expose 3 functions for developers to fill up: Model, Update and Signal Generator.

```
Syntax
| value | \(V, W ::= i | c | x | \lambda x.M | <V, W> | i\) |
|-------|--------------------------------------|
| program | \(M, N ::= V | MN | op(M)\) |
| \(n \in \mathbb{R} \cup \mathbb{B}\) |
| \(x \in \text{Var}\) |
| \(i \in \text{Input}\) |
```

Type System
```
\(\tau :: \text{unit} | \text{number} | \text{bool} | \tau \rightarrow \tau'\)
\(\sigma :: \text{signal } \tau | \tau \rightarrow \sigma | \sigma \rightarrow \sigma'\)
\(\eta :: \tau | \sigma\)
```

Type Judgments
```
UNIT \(f \mapsto ()::\text{unit}\) 
NUMBER \(f \mapsto n::\text{number}\) 
BOOL \(f \mapsto n::\text{bool}\) 
LET \(f \mapsto e_1;\eta :: \tau \rightarrow \sigma\) 
INPUT \(f \mapsto (_) :: \text{signal } \tau\)
```

VAR \(f \mapsto x::\text{var}\) 
LAMBDA \(f \mapsto \lambda x.\eta :: \tau\rightarrow \sigma\) 
LET \(f \mapsto x::\text{var}\) 
APPLY \(f \mapsto e_1;e_2::\tau\rightarrow \sigma\) 
FOLD \(f \mapsto \text{fold}_e e_1;e_2;e_3::\sigma\rightarrow \sigma'\)