Hailstorm : A Statically-Typed, Purely Functional Language for IoT Applications

Downloaded from: https://research.chalmers.se, 2022-05-19 07:27 UTC

Citation for the original published paper (version of record):

N.B. When citing this work, cite the original published paper.
Hailstorm : A Statically-Typed, Purely Functional Language for IoT Applications

Abhiroop Sarkar
sarkara@chalmers.se
Chalmers University
Gothenburg, Sweden

Mary Sheeran
mary.sheeran@chalmers.se
Chalmers University
Gothenburg, Sweden

ABSTRACT

With the growing ubiquity of Internet of Things (IoT), more complex logic is being programmed on resource-constrained IoT devices, almost exclusively using the C programming language. While C provides low-level control over memory, it lacks a number of high-level programming abstractions such as higher-order functions, polymorphism, strong static typing, memory safety, and automatic memory management.

We present Hailstorm, a statically-typed, purely functional programming language that attempts to address the above problem. It is a high-level programming language with a strict typing discipline. It supports features like higher-order functions, tail-recursion, and automatic memory management, to program IoT devices in a declarative manner. Applications running on these devices tend to be heavily dominated by I/O. Hailstorm tracks side effects like I/O in its type system using resource types. This choice allowed us to explore the design of a purely functional standalone language, in an area where it is more common to embed a functional core in an imperative shell. The language borrows the combinators of arrowized FRP, but has discrete-time semantics. The design of the full set of combinators is work in progress, driven by examples.

So far, we have evaluated Hailstorm by writing standard examples from the literature (earthquake detection, a railway crossing system and various other clocked systems), and also running examples on the GRiSP embedded systems board, through generation of Erlang.

CCS CONCEPTS

• Software and its engineering → Compilers; Domain specific languages; • Computer systems organization → Sensors and actuators; Embedded software.

KEYWORDS

functional programming, IoT, compilers, embedded systems

ACM Reference Format:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.


1 INTRODUCTION

As the density of IoT devices and diversity in IoT applications continue to increase, both industry and academia are moving towards decentralized system architectures like edge computing [38]. In edge computation, devices such as sensors and client applications are provided greater computational power, rather than pushing the data to a backend cloud service for computation. This results in improved response time and saves network bandwidth and energy consumption [50]. In a growing number of applications such as aeronautics and automated vehicles, the real-time computation is more robust and responsive if the edge devices are compute capable.

In a more traditional centralized architecture, the sensors and actuators have little logic in them; they rather act as data relaying services. In such cases, the firmware on the devices is relatively simple and programmed almost exclusively using the C programming language. However with the growing popularity of edge computation, more complex logic is moving to the edge IoT devices. In such circumstances, programs written using C tend to be verbose, error-prone and unsafe [17, 27]. Additionally, IoT applications written in low-level languages are highly prone to security vulnerabilities [7, 58].

Hailstorm is a domain-specific language that attempts to address these issues by bringing ideas and abstractions from the functional and reactive programming communities to programming IoT applications. Hailstorm is a pure, statically-typed functional programming language. Unlike impure functional languages like ML and Scheme, Hailstorm restricts arbitrary side-effects and makes dataflow explicit. The purity and static typing features of the language, aside from providing a preliminary static-analysis tool, provide an essential foundation for embedding advanced language-based security techniques [54] in the future.

The programming model of Hailstorm draws inspiration from the extensive work on Functional Reactive Programming (FRP) [18]. FRP provides an interface to write reactive programs such as graphic animations using (1) continuous time-varying values called Behaviours and (2) discrete values called Events. The original formulation of FRP suffered from a number of shortcomings such as space-leaks [41] and a restrictive form of stream based I/O.

A later FRP formulation, arrowized FRP [46], fixed space leaks, and in more recent work Winograd-Cort et al. introduced a notion of resource types [66] to overcome the shortcomings of the stream based I/O model. The work on resource types is a library in Haskell, and is not suitable to run directly on resource-constrained hardware. Hailstorm uses resource types to uniquely identify each I/O
resource. It treats each resource as a signal function to track its lifetime, and prevents resource duplication through the type system. Hailstorm currently has a simple discrete time semantics, though we hope to explore extensions later.

A Hailstorm program is compiled to a dataflow graph, which is executed synchronously. The core of the language is a pure call-by-value implementation of the lambda calculus. The synchronous language of arrowized-FRP provides a minimal set of combinators to which the pure core constructs of Hailstorm can be raised. This language of arrows then enforces a purely functional way to interact with I/O, using resource types.

Hailstorm, in its current version, is a work-in-progress compiler which does not address the reliability concerns associated with node and communication failures plaguing edge devices [58]. We discuss the future extensions of the language to tackle both reliability and security concerns in Section 7. We summarize the contributions of Hailstorm as follows:

- **A statically-typed purely functional language for IoT applications.** Hailstorm provides a tailored, purely functional alternative to the current state of programming resource constrained IoT devices.
- **Resource Types based I/O.** Hailstorm builds on Winograd-Cort et al’s work to provide the semantics and implementation of an alternate model of I/O for pure functional languages using resource types - which fits the streaming programming model of IoT applications. (Section 4.1)
- **Discrete time implementation** Hailstorm uses the combinators of arrowized FRP in a discrete time setting (Section 3).
- **An implementation of the Hailstorm language.** We implement Hailstorm as a standalone compiler, with Erlang and LLVM backends. We have run case studies on the GRISt embedded system boards [60], to evaluate the features of the language. (Section 4). The compiler implementation and the examples presented in the paper are made publicly available¹.

## 2 LANGUAGE OVERVIEW

In this section we demonstrate the core features and syntax of Hailstorm using running examples. We start with a simple pure function that computes the $n^{th}$ Fibonacci number.

```python
def main : Int = fib 6

def fib : (Int -> Int)
  = fib_help 0 1

def fib_help (a : Int) (b : Int) (n : Int) : Int
  = if n == 0
      then a
      else fib_help (a + b) a (n - 1)
```

The simple program above, besides showing the ML-like syntax of Hailstorm, demonstrates some features like (1) higher-order functions (2) recursion (3) partial application (4) tail-call optimization and (5) static typing. All top-level functions in a Hailstorm program have to be annotated with the types of the arguments and the return type, which currently allows only monomorphic types. However certain built-in combinators supported by the language are polymorphic which will be discussed in the following section.

The pure core of the language only allows writing pure functions which have no form of interactions with the outside world. To introduce I/O and other side effects, we need to describe the concept of a signal function.

### 2.1 Signal Functions

A fundamental concept underlying the programming model of Hailstorm is that of a Signal Function. Signal Functions, derived from the work on arrowized-FRP [46], are functions that always accepts an input and always returns an output.

Signal functions are analogous to the nodes of a dataflow graph. Signal functions operate on signals which do not have any concrete representation in the language. A signal denotes a discrete value at a give point of time. Nilsson et al [46] use the electric circuit analogy: a signal corresponds to a wire and the current flowing through it, while signal functions correspond to circuit components. An important distinction between Hailstorm and both classic and arrowized-FRP is that signals are always treated as discrete entities in Hailstorm unlike the continuous semantics enforced by FRP.

To create larger programs Hailstorm provides a number of built-in combinators to compose signal functions. These combinators are drawn from the Arrow framework [31] which is a generalization of monads. Arrows allow structuring programs in a point-free style, while providing mathematical laws for composition. We start by presenting some of the core Hailstorm combinators² and their types for composing signal functions.

```
mapSignal# : (a -> b) -> SF a b
>>> : SF a b -> SF b c -> SF a c
&&& : SF a b -> SF a c -> SF b c
<<< : SF a b -> SF c d -> SF (a, c) (b, d)
```

Some of the built-in combinators in Hailstorm are polymorphic and the type parameters $a$, $b$ and $c$ represent the polymorphic types. `mapSignal#` is the core combinator which lifts a pure Hailstorm function to the synchronous language of arrows, as a signal function (See Fig 1).

Hailstorm then provides the rest of the built-in combinators to compose signal functions while satisfying nine arrow laws [39]. One of the advantages of having a pure functional language is that such laws can be freely used by an optimizing compiler to aggressively inline and produce optimized code. The semantics of composing signals with the arrow combinators is visually depicted in Fig 1.

¹https://abhiroop.github.io/ppdp-2020-artifact.zip
²Non-symbolic built-in combinators & driver functions in Hailstorm end with #
A natural extension to signal functions is using them to interact with I/O, which we discuss in the next section.

We adopt this model in Hailstorm. The type parameter \( \emptyset \) represents a void type. There are no values in the language which inhabit the \( \emptyset \) type. The \( \emptyset \) type always appears within a signal function. So, an example like \( \text{sensor} : \text{SF} (\emptyset) \text{Int} \) - this represents an action which when called, produces an integer.

One of the key aspects of designing a pure functional language is this distinction between an expression that returns a value and an action. When an action (like \( \text{sensor} : \text{SF} (\emptyset) \text{Int} \)) is evaluated, it returns a representation of that function call rather than actually executing the call itself. This distinction is the key to equational reasoning in a purely functional program. In the absence of such a distinction the following two expressions are no longer equivalent although they represent the same programs.

![Figure 1: Arrow combinators for signal functions](image)

Now where do signals actually come from? To answer this question - a natural extension to signal functions is using them to interact with I/O, which we discuss in the next section.

### 2.2 I/O

Hailstorm adopts a streaming programming model, where an effectful program is constructed by composing various signal functions in the program. The final program is embedded in a stream of input flowing in and the program transforms that into the output.

Figure 2: A Hailstorm program interacting with the real world

While this model of I/O adapts well with a pure functional language, it is reminiscent of the now abandoned stream-based I/O interface in Haskell. In Haskell’s early stream-based I/O model, the type of the main function was \([\text{Response}] \rightarrow [\text{Request}]\). The major problems with this model are:

- All forms of I/O are restricted to happen at the main function leading to non-modular programs, especially in case of applications running on IoT devices where I/O functions dominate the majority of the program.
- It is non-extensible as the concrete types of Request and Response need to be altered every time any new I/O facility has to be added [48].
- There is also no clear mapping between an individual Request and its Response [48].

We adopt this model in Hailstorm. The type parameter \( \emptyset \) represents a void type. There are no values in the language which inhabit the \( \emptyset \) type. The \( \emptyset \) type always appears within a signal function. So, an example like \( \text{sensor} : \text{SF} (\emptyset) \text{Int} \) - this represents an action which when called, produces an integer.

One of the key aspects of designing a pure functional language is this distinction between an expression that returns a value and an action. When an action (like \( \text{sensor} : \text{SF} (\emptyset) \text{Int} \)) is evaluated, it returns a representation of that function call rather than actually executing the call itself. This distinction is the key to equational reasoning in a purely functional program. In the absence of such a distinction the following two expressions are no longer equivalent although they represent the same programs.

![Figure 1: Arrow combinators for signal functions](image)

Now where do signals actually come from? To answer this question - a natural extension to signal functions is using them to interact with I/O, which we discuss in the next section.

We adopt this model in Hailstorm. The type parameter \( \emptyset \) represents a void type. There are no values in the language which inhabit the \( \emptyset \) type. The \( \emptyset \) type always appears within a signal function. So, an example like \( \text{sensor} : \text{SF} (\emptyset) \text{Int} \) - this represents an action which when called, produces an integer.

One of the key aspects of designing a pure functional language is this distinction between an expression that returns a value and an action. When an action (like \( \text{sensor} : \text{SF} (\emptyset) \text{Int} \)) is evaluated, it returns a representation of that function call rather than actually executing the call itself. This distinction is the key to equational reasoning in a purely functional program. In the absence of such a distinction the following two expressions are no longer equivalent although they represent the same programs.

![Figure 1: Arrow combinators for signal functions](image)

Now where do signals actually come from? To answer this question - a natural extension to signal functions is using them to interact with I/O, which we discuss in the next section.

We adopt this model in Hailstorm. The type parameter \( \emptyset \) represents a void type. There are no values in the language which inhabit the \( \emptyset \) type. The \( \emptyset \) type always appears within a signal function. So, an example like \( \text{sensor} : \text{SF} (\emptyset) \text{Int} \) - this represents an action which when called, produces an integer.

One of the key aspects of designing a pure functional language is this distinction between an expression that returns a value and an action. When an action (like \( \text{sensor} : \text{SF} (\emptyset) \text{Int} \)) is evaluated, it returns a representation of that function call rather than actually executing the call itself. This distinction is the key to equational reasoning in a purely functional program. In the absence of such a distinction the following two expressions are no longer equivalent although they represent the same programs.

![Figure 1: Arrow combinators for signal functions](image)

Now where do signals actually come from? To answer this question - a natural extension to signal functions is using them to interact with I/O, which we discuss in the next section.

We adopt this model in Hailstorm. The type parameter \( \emptyset \) represents a void type. There are no values in the language which inhabit the \( \emptyset \) type. The \( \emptyset \) type always appears within a signal function. So, an example like \( \text{sensor} : \text{SF} (\emptyset) \text{Int} \) - this represents an action which when called, produces an integer.

One of the key aspects of designing a pure functional language is this distinction between an expression that returns a value and an action. When an action (like \( \text{sensor} : \text{SF} (\emptyset) \text{Int} \)) is evaluated, it returns a representation of that function call rather than actually executing the call itself. This distinction is the key to equational reasoning in a purely functional program. In the absence of such a distinction the following two expressions are no longer equivalent although they represent the same programs.

![Figure 1: Arrow combinators for signal functions](image)

Now where do signals actually come from? To answer this question - a natural extension to signal functions is using them to interact with I/O, which we discuss in the next section.
Hailstorm function to a signal function without any effectful operations, and as a result the resource type is Empty. The combinators \(\gg\gg\gg\) and \(***\) compose signal functions, and result in a disjoint-union of the two resources types. This type-level disjoint union prevents us from copying the same resource using any of these combinators. So in Hailstorm if we try this,

\[
def foo : SF (S \cup S) () (Int, Int) = sensor \&\& sensor
\]

we currently get the following upon compilation:

Type-Checking Error:
Error in "foo":
Cannot compose resources : S S containing same resource
Encountered in
sensor \&\& sensor

The type rules associated with composing Hailstorm combinators and their operational semantics are presented formally in Section 3.2 and 3.3 respectively.

2.2.1 Example of performing I/O. We can distinguish the read and write interface of a resource using two separate resource types. For example, to repeatedly blink an LED we need two APIs - (1) to read its status (2) to write to it. The drivers for these two functions have the following types:

\[
\text{readLed}\# : SF R () \text{Int} \\
\text{writeLed}\# : SF W \text{Int} ()
\]

We use the integer 1 to represent light ON status and 0 for OFF. The program for blinking the LED would be:

\[
def main : SF (R \cup W) () () = \text{readLed}\# \gg\gg\gg \text{mapSignal}\# \text{flip} \gg\gg\gg \text{writeLed}\#
\]

\[
def \text{flip} (s : \text{Int}) : \text{Int} = \\
\quad \text{if} \ (s == 0) \ \text{then} \ 1 \ \text{else} \ 0
\]

The above program runs the function \(\text{main}\) infinitely. It is possible to adjust the rate at which we want to run this program, discussed later in Section 2.5. This treatment of I/O as signal functions has the limitation that each device (as well as their various APIs) has to be statically encoded as a resource type in the program.

2.3 State

Hailstorm supports stateful operations on signals using the \(\text{loop}\#\) combinator.

\[
\text{loop}\# : c \rightarrow SF \text{Empty} (a, c) (b, c) \rightarrow SF \text{Empty} a b
\]

The type of the \(\text{loop}\#\) combinator is slightly different from the type provided by the \texttt{ArrowLoop} typeclass in Haskell, in that it allows initializing the state type variable \(c\). The internal body of the signal function encapsulates a polymorphic state entity. This entity is repeatedly fed back as an additional input, upon completion of a whole step of signal processing by the entire dataflow graph. Fig 3 represents the combinator visually.

The \(\text{loop}\#\) combinator can be used to construct the \(\text{delay}\) function as found in synchronous languages like Lustre [26], for encoding state.

\[
def \text{delay} (x : \text{Int}) : SF \text{Empty} \text{Int} \text{Int} = \text{loop}\# x (\text{mapSignal}\# \text{swap})
\]

\[
def \text{swap} (a : \text{Int}, s : \text{Int}) : (\text{Int}, \text{Int}) = (s, a)
\]

2.4 A sample application

We now demonstrate the use of the Hailstorm combinators in a sample application. The application that we choose is a simplified version of an earthquake detection algorithm [65] which was first used by Mainland et al. to demonstrate their domain specific language for wireless sensor networks [42]. The figure below shows the core dataflow graph of the algorithm.

The exponentially weighted moving average (EWMA) component above is a stateful element. We assume the \texttt{getSample} input function is a wrapper around a seismometer providing readings of discrete samples. At the rightmost end, the Detection Event would be another stateful entity which would include some form of an edge detector. The entire program for the earthquake detection is given in Fig 5.

The function \(\text{edge}\) in Fig 5 is a stateful edge detector which generates an action if a boolean signal changes from \texttt{False} to \texttt{True}. To program this, we use an imaginary actuator (like an LED) in a GRISP board, which would glow red once if the input to it is 1, signalling danger, and otherwise stay green signalling no earthquake.

2.5 Sampling rate

The combinators introduced so far execute \textit{instantaneously} using a logical clock. In Hailstorm, one logical time step includes the following actions, in sequence -

- accepting a discrete sample of data from each of its connected input devices
- passing the discrete sample through the dataflow graph
- finally passing a discrete value to the responsible actuator
This helps us establish a relation between the wall clock time and (in seconds) at which we wish to set the period of sampling input.

resents RESET and 3 represents STOP.

Int accepts an input stream of

A program returning a signal function continuously loops around, streaming in input and executing the above steps, at the speed it takes for the instructions to execute. However under most circumstances we might wish to set a slower rate for the program. The rate# combinator is used for that purpose.

The first argument to rate# is the length of the wall clock time (in seconds) at which we wish to set the period of sampling input. This helps us establish a relation between the wall clock time and Hailstorm’s logical clock. We demonstrate the utility of the rate# combinator using a Stopwatch example.

2.5.1 Stopwatch. We program a hypothetical stopwatch which accepts an input stream of Ints where 1 represents START, 2 represents RESET and 3 represents STOP.

Figure 5: The earthquake detection algorithm

Figure 6: A switch activating only the top signal function

2.5.2 Limitation. In the current implementation of Hailstorm, an operation like \((\text{rate} t_2 (\text{rate} t_1 \text{sf}))\) would result in setting the final sampling rate as \(t_2\), overwriting the value of \(t_1\). In the programs presented here, we use a single clock, and hence a single sampling rate. Libraries like Yampa [14] provide combinators like \(\text{delay} \colon \text{Time} \to \text{a} \to \text{SF a a}\) which allow oversampling for dealing with multiple discrete sampling rates. As future work, we hope to adopt oversampling operators for communication among signal functions with different sampling rates.

2.6 Switches

A Hailstorm dataflow graph allows a form of dynamic, data-driven switching within the graph. It accomplishes this using the switch# combinator:

The first argument to switch# accepts a signal function whose output data is used to switch between the various signal functions. The strict typing of Hailstorm restricts the branches of the switch to be of the same type, including the resource type. The switching dataflow is visually presented in Fig 6.

The Arrow framework provides more useful combinators based on the ArrowChoice typeclass which are currently absent from Hailstorm.
- The switch# combinator is an experiment to describe expressions of the form:

<table>
<thead>
<tr>
<th>Switch# input_signal_function</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{\texttt{\textunderscore val \Rightarrow if &lt;condition&gt; on val}} )</td>
</tr>
<tr>
<td>( \text{then SF1} )</td>
</tr>
<tr>
<td>( \text{else if &lt;condition&gt; on val}} )</td>
</tr>
<tr>
<td>( \text{then SF2} )</td>
</tr>
<tr>
<td>( \text{else \ldots} )</td>
</tr>
</tbody>
</table>

The combinator currently supports expression only of the above form. Thus, we currently do not allow more general functions of type \( (b \rightarrow SF \ b \rightarrow c) \) as the second parameter to switch#, and so avoid problems with possibly undefined runtime behaviour. However, there is no check in the compiler to enforce this restriction. As future work we hope to adopt the guards syntax of Haskell to represent the second parameter as a collection of boolean clauses and their corresponding actions.

3 SYNTAX, SEMANTICS AND TYPES

In the previous sections, we have given an informal treatment to the most important syntactic parts of Hailstorm, for IoT applications, and described the programming model using them. In this section, we present the core syntax, type rules and operational semantics of the main parts of Hailstorm.

3.1 Syntax

The set of types in the source language is given by the following grammar:

\[ r ::= () | \text{Int} | \text{Float} | \text{Bool} | \tau_1 \rightarrow \tau_2 | \tau_1 \Rightarrow \tau_2 | (\tau_1, \tau_2) \]

The type \( \tau_1 \Rightarrow \tau_2 \) represents a signal function type from \( a \) to \( b \) with the resource type \( r \) of \( r \) a \( b \).

The abstract syntax of the core expressions of Hailstorm is given by the following grammar. The meta-variable \( x \in \text{Var} \) ranges over variables of the source language. Additionally we let \( i \in \mathbb{Z} \) and \( f \in \mathbb{R} \). We use \( e \) and \( e_{sf} \) separately to denote ordinary expressions and arrow based signal function expressions respectively.

\[
e ::= x | i | f | \text{True} | \text{False} | i_1 \ | \text{binop}_1 \ i_2 | f_2 \ | \text{binop}_f \ f_2
\]

\[
| e_1 \ 	ext{rel} \ e_2 | \text{if} \ e_1 \ \text{then} \ e_2 \ \text{else} \ e_3 | \text{let} \ x = e_1 \ \text{in} \ e_2 | (e_1, e_2) | \text{fst}# \ e | \text{snd}# \ e
\]

\[
e_{sf} ::= \text{mapSignal#} (\text{lx.e})(e_{sf1} \ggg e_{sf2})
\]

\[
| e_{sf1} \ & & e_{sf2} (e_{sf2} \ & & ) e_{sf2} \ | e_{sf2} \ | e_{sf2} \ | e_{sf2} \ | e_{sf2} \ | \text{loop}# \ e_1 \ e_2 \ | \text{switch#} \ e_1 (\text{lx.e})| \text{read#}| \text{write#}
\]

\[
\text{binop}_1 ::= + \ | - \ | *
\]

\[
\text{binop}_f ::= . \ | \ - \ | * \ | / \ | \text{rel} \ ::= > \ | < \ | >= \ | == \ | ===
\]

In the grammar above we describe two primitives for I/O called read# and write#. In practise, as Hailstorm deals with a number of I/O drivers there exists a variety of I/O primitives with varied parameters and return types. However for the purpose of presenting the operational semantics, we abstract away the complexity of the drivers and use the abstracted read# and write# to describe the semantics in Section 3.3.

3.2 Type rules

Hailstorm uses a fairly standard set of type rules except for the notion of resource types. The typing context of Hailstorm employs dual contexts, in that it maintains (1) \( \Gamma \cdot \) a finite map from variables to their types and (2) \( \Delta \cdot \) a finite set which tracks all the I/O resources connected to the program.

\[
\Delta;\Gamma ::= \cdot | \Gamma, x : \tau
\]

An empty context is given by \( \cdot \). Additionally \( \text{dom}(\Gamma) \) provides the set of variables bound by a typing context.

In Fig 7, we show the most relevant type rules concerning signal functions and their composition. The remaining expressions follow standard set of type rules which is provided in its entirety in the extended version of this paper [55].

Looking at the rule \( T\text{-mapSignal} \), a signal function such as \( r_1 \Rightarrow \emptyset \) \( r_2 \) denotes the result of applying \( \text{mapSignal#} \) to a pure function. This results in an expression with an empty resource type denoted by \( \emptyset \). A missing rule is the introduction of a new resource type in the resource type context \( \Delta \). The resource type context is an append-only store and a new resource is introduced using the keyword resource. It can be defined using this simple reduction semantics

\[
\Delta;\Gamma \vdash \text{resource } r \rightsquigarrow \Delta \cup r;\Gamma
\]

where \( \rightsquigarrow \) denotes one step of reduction which occurs at compile-time. The rules such as \( \text{T-Compose} \), \( \text{T-Fanout} \), \( \text{T-Combine} \), \( \text{T-Switch} \) apply a type-level disjoint union to prevent resource duplication.

3.3 Big-step Operational Semantics

In this section, we provide a big-step operational semantics of our implementation of the Hailstorm language, by mapping the meaning of the terms to the lambda calculus. We begin by defining the values in lambda calculus that cannot be further reduced:

\[
V ::= i | f | \lambda x. E
\]

\( i \) and \( f \) represent integer and float constants respectively. We use \( n \) to represent variables and the last term denotes a lambda expression. The syntax that we use for defining our judgements is of the form:

\[
s_1 \vdash M \Downarrow V_1, s_2
\]

The variables \( s_1, s_2 \) are finite partial functions from variables \( n \) to their bound values \( V_1 \in V \). In case a variable \( n \) is unbound and \( s \) is called with that argument it returns \( \emptyset \). The above judgement is read as starting at state \( s_1 \) and evaluating the term \( M \) results in the irreducible value \( V_1 \in V \) while setting the final state to \( s_2 \).

The first judgement essential for our semantics is of the form:

\[
s \vdash V \Downarrow V, s
\]

which means that the values \( V \) cannot be reduced further.

We use a shorthand notation \( p(n, s) \) to signify lookup the variable \( n \) in \( s \). Additionally, to make the semantics more compact we
They do not belong to \( s \) transition from composing signal functions with different sampling rates, the state use pairs \( <a, b> \) and their first and second projections, \( \text{fst}, \text{snd} \). They do not belong to \( V \) but it is possible to represent all three of them using plain lambdas and function application - shown in the extended version of this paper [55].

In Fig 8, we show the most relevant big-step operational semantics concerning signal function based combinators. The remaining expressions have standard semantics and the complete rule set is provided in the extended paper [55]. In the rule \( \text{eval-Read} \), we use \( \Psi \) to store the sampling rate. In our current implementation, when composing signal functions with different sampling rates, the state transition from \( s_2 \) to \( s_3 \) overwrites the first sampling rate.

In \( \text{eval-Loop-Init} \) and \( \text{eval-Loop} \), the subscript \( n \) represents a variable name that is used as a key, in the global state map \( s \), to identify each individual state.

For the rule \( \text{eval-Switch} \), \( \sigma_b[\lambda_c.E_1, \lambda_c.E_2, ... \lambda_c.E_n] \) represents a conditional expression that uses the value of \( b \) i.e. \((V_1, a)\) to choose one of the several branches - \( \lambda_c.E_i \) - and then supplies \((V_1, a)\) again to the selected branch to actually generate a value of the stream.

Of special interest in Fig 8 are the rules \( \text{eval-Read} \) and \( \text{eval-Write} \). We need to extend our lambda calculus based abstract machine with the operations, \( \text{read} \) and \( \text{write} \), to allow any form of I/O. The effectful operations, \( \text{read} \) and \( \text{write} \), are guarded by \( \lambda s \) to prevent
any further evaluation, and as a result are treated as values. This method is essential to ensure the purity of the language - by treating effectful operations as values.

The program undergoes a partial evaluation transformation which evaluates the entire program to get rid of all the λs guarding the read operations. Given the expression \( \lambda x. \text{read} \) the compiler supplies a compile time token of type \( \odot \) which removes the lambda and exposes the effectful function \( \text{read} \). The partially evaluated program is then prepared to conduct I/O. This approach is detailed further in Section 4.1.

The big-step semantics of the language shows its evaluation strategy. However to understand the streaming, infinite nature of an effectful Hailstorm program we need an additional semantic rule. A Hailstorm function definition is itself an expression and a program is made of a list of such functions,

\[
\text{Program} ::= \text{main} : [\text{Function}] \\
\text{Function} ::= e | e_1 e_2
\]

Each Hailstorm program compulsorily has a \text{main} function. After the entire program is partially evaluated (described in Section 4.1) and given that the \text{main} function causes a side effect (denoted by \( \odot \) below), we have the following rule:

\[
\frac{\text{s}_1 \vdash \text{main} \odot(), \text{s}_2}{\text{s}_1 \vdash \text{main} \odot(), \text{main}, \text{s}_2} \quad \text{(eval-Main)}
\]

The eval-Main rule demonstrates the streaming and infinite nature of a Hailstorm program when the main function is a signal function itself. After causing a side effect, it calls itself again and continues the stream of effects while evaluating the program using the semantics of Fig 8.

4 IMPLEMENTATION

Here we describe an implementation of the Hailstorm language and programming model presented in the previous sections. We implement the language as a compiler - the Hailstorm compiler - whose compilation architecture is described below.

The compiler pipeline starts by parsing a Hailstorm source file and desugaring small syntactic conveniences provided to reduce code size (such as case expressions). After desugaring, the Hailstorm AST constituted of the grammar described in Section 3.1 is generated. Next, typechecking of the AST using the type rules of Fig 7 is done. After a program is typechecked, it is transformed into the Hailstorm core language called L1.

The L1 language is an enriched version of the simply typed lambda calculus (STLC) with only nine constructors. Unlike STLC, L1 supports recursion by calling the name of a global function. It is currently incapable of recursion using a \text{let} binding. L1 has a simpler type system than the Hailstorm source, as it erases the notion of resource types - which are exclusively used during type checking.

The L1 language being considerably simpler, forms a sufficient foundation for running correctness preserving optimization passes. L1 attempts to get rid of all closures with free variables, as they are primarily responsible for dynamic memory allocation. Additionally, it attempts to inline expressions (primarily partial applications) to further reduce both stack and heap memory allocation. The optimization passes are described in further detail in Section 4.2.

The final optimization pass in L1 is a partial evaluation pass which specializes the program to convert it into an effectful program. Before this pass, all I/O inducing functions are treated as values, by guarding them inside a \( \lambda \) abstraction. This pass evaluates those expressions by passing a compile-time token and turning the L1 non-effectful program into an effectful one. This pass is discussed in further detail in Section 4.1.

Finally, the effectful Hailstorm program gets connected to one of its backends. Recursion is individually handled in each of the backends by using a global symbol table. We currently support an LLVM backend [36] and a BEAM backend [4].

4.1 I/O

The I/O handling mechanism of the Hailstorm compiler is the essential component in making the language pure. A pure functional language allows a programmer to equationally reason about their code. The entire program is written as an order independent set of equations. However, to perform I/O in a programming language, it is necessary to (1) enforce an order on the I/O interactions, as they involve chronological effects visible to the user and (2) interact with the real world and actually perform an effect.

To solve (1) we use the eval-App rule given in Fig 8. Hailstorm, being a call-by-value implementation of the lambda calculus, follows ordering in function application by always evaluating the function argument before passing it to the function. This semantics of function application allows us to introduce some form of ordering to the equations.

To solve (2) we have extended our pure lambda calculus core to involve effectful operations like \text{read} and \text{write}. Again, as observed in the semantic rules, eval-Read and eval-Write from Fig 8, the effectful operations are guarded by \( \lambda \) abstractions to treat them as values, rather than operations causing side-effects. This allows us to freely inline or apply any other optimization passes while preserving the correctness of the code. However, to finally perform the side effect, we need to resolve this lambda abstraction at compile
time. We do this by partially evaluating [33] our program. Below we describe the semantics of the partial evaluation step.

A Hailstorm program is always enforced (by the typechecker in our implementation) to contain a main function. We shall address the case of an effectful program i.e a main function whose return type is an effectful signal function such as SF $r \rightarrow \tau$. The $\tau$ type in the input and output parameters reflect that this function reads from a real world source like a sensor and causes an effect such as moving an actuator. The partial application pass is only fired when the program contains the $\rightarrow$ type in either one of its signal function parameters.

Let us name the main function above with the return type of SF $r \rightarrow \tau$ as main$_{sf}$. Now main$_{sf}$ is itself a function, embedded in a stream of input and transforming the input to an output, causing an effect. As it is a function we can write: main$_{sf} \equiv \lambda \tau.E$.

Additionally in our implementation, after typechecking, a signal function type $\tau_1 \Rightarrow \tau_2$ is reduced to a plain arrow type $\tau_1 \rightarrow \tau_2$. Now, using the two aforementioned definitions, we can write the following reduction semantics:

$$\text{main}_{sf} = \lambda \tau.E \quad \text{main}_{sf} : \tau \Rightarrow \tau \quad \tau_1 \Rightarrow \tau_2 \Rightarrow \tau_1 \rightarrow \tau_2$$

where $\Rightarrow$ denotes one step of reduction and $\Delta$ an arbitrary compile time token of type $\tau$. The final step, which produces the expression $E[\theta : \tau (\Delta)]$, is the partial evaluation step. We demonstrate this reduction semantics in action using an example:

$$\text{read} : \text{SF STDIN} \rightarrow \text{Int}$$
$$\text{write} : \text{SF STDOUT} \rightarrow \text{Int}$$

We use the eval-Read, eval-Write and eval-Compose rules to translate the main function above to

$$\text{main} = \lambda (\text{ly}. \text{write} \ y) ((\text{lx}. \text{read} \ t))$$

Given the above definition of main, we can apply the reduction rule eval-Partial and eval-App to get the following.

$$\lambda (\text{ly}. \text{write} \ y) ((\text{lx}. \text{read} \ t)) \Rightarrow (\lambda y. \text{write} \ y) (\text{read})$$

Now the program is ready to create a side effect as the read function is no longer guarded by a $\lambda$ abstraction. The eval-App rule guarantees that read is evaluated first and only then the value is fed to write owing to the call-by-value semantics of Hailstorm.

4.1.1 Limitation. The Hailstorm type system doesn’t prevent a programmer from writing write# >>> read#. The type of such a program would be SF (STDOUT U STDIN) Int Int. As the types do not reflect the $\Rightarrow$ type, the partial evaluation pass is not fired and the program simply generates an unevaluated closure - which is an expected result given the meaningless nature of the program. However, the type would allow composing it with other pure functions and producing bad behaviour if those programs are connected to meaningful I/O functions. This is simply solved by adding the following type rule:

$$\Gamma \vdash \tau_1 \Rightarrow \tau_2 \quad \tau \in \Lambda \quad \tau \neq \emptyset \quad \Delta \vdash \tau_1 = () \land \tau_2 = () \quad (\text{T-Unsafe})$$

4.2 Optimizations

4.2.1 Lambda Lifting. Hailstorm, being a functional language, supports higher order functions (HOFs). HOFs frequently capture free variables which survive the scope of a function call. For example:

```haskell
def addFive (nr : Int) : Int =
  let x = 5 in
  let addx = \(y : \text{Int}) \Rightarrow x + y in
  addx nr
```

Our implementation treats HOFs as closures which are capable of capturing an environment by allocating the environment on the heap. In line no. 3 above, the value of the variable $x$ is heap allocated. However, in resource constrained devices heap memory allocation, is highly restrictive and any language targeting such devices should attempt to minimize allocation.

Hailstorm applies a lambda-lifting [32] transformation to address this. Lambda-lifting lifts a lambda expression with free variables to a top-level function and then updates related call sites with a call to the top-level function. The free variables then act as arguments to the function. This effectively allocates them on the stack (or registers). Owing to our restricted language and the lack of polymorphism, our algorithm is less sophisticated than the original algorithm devised by Johnsson. We describe the operational semantics of our algorithm in the lambda-lifting rule below.

We use a slightly different notation from Section 3.3 here. $P_n$ is used to describe the entire program with its collection of top level functions. We identify a modified program with $P_n[G(x).E]$ to mean a new program with an additional global function $G$ which accepts an argument $x$ and returns an expression $E$. Finally the $fv$ function is used find the set of free variables in a Hailstorm expression.

$$P_1 \vdash \exp \Downarrow \lambda x. E, \ P_2 \quad \text{fv}(\lambda x. E) \neq \emptyset$$

$$f v(\lambda x. E) = \{i_1, ..., i_n\} \quad P_2 \vdash \lambda x. E \Downarrow E', P_3[G(i_1, ..., i_n, x). E] \quad P_1 \vdash \exp \Downarrow E', P_3[G(i_1, ..., i_n, x). E]$$

The above rule returns a modified expression $E'$ which consists of a function call to $G(i_1, ..., i_n, x). E$ with the free variables $i_1, ... , i_n$ as arguments. This rule is repeatedly run on local lambda expressions until none has any free variables. It transforms the program foo above to:

```haskell
def addFive' (nr : Int) = x + y
```

4.2.2 Inlining. The inlining transformation works in tandem with the lambda lifter to reduce memory allocations. Inlining a lambda calculus based language reduces to plain $\beta$ reduction, i.e $(\lambda x. M) E \Rightarrow M[x \mapsto E]$. Inlining subsumes optimization passes like copy-propagation in a functional language. Our prototype inliner is relatively conservative in that (1) it doesn’t attempt to inline recursive functions and (2) it doesn’t attempt inter-function inlining.

However, it attempts to cooperate with the lambda-lifter to remove all possible sites of partial applications, which are also heap allocated, to minimize memory allocation. The program shown in the previous section, undergoes the cycle in Fig 10
The lambda lifting pass produces a partial application which is further inlined to a single function call where the arguments can be passed using registers. We show the generated LLVM code before and after the optimization passes are run in the extended version of this paper [55]. We show there the absence of any calls to malloc in the optimized version of the code. Our inliner doesn’t employ any novel techniques but it still has to deal with engineering challenges like dealing with name capture [5], which it solves using techniques from the GHC inliner [34].

4.3 Code Generation

The Hailstorm compiler is designed as a linear pass through a tower of interpreters which compile away high level features, in the tradition of Reynolds [51]. Here, we show the final code generation for two of the more interesting combinators using C-like notation.

4.3.1 loop#. The loop# combinator models a traditional Mealy machine whose output depends on the input as well as the current state of the machine. In the following we see the code generated for the delay combinator described in Section 2.3 which is itself described using loop#.

In Erlang, the global variable of foo_state is modeled recursively using a global state map, which is updated on every time step. In the LLVM backend, the translation is very similar to the C notation shown above.

4.3.2 switch#. We show an example of the switch# combinator when dealing with stateful branches

Figure 11: Code generation for switch#

The second parameter in the switch# type -(b→SF b c) - acts like a macro which is unfolded into an if-then-else expression in the L1 core language. The core language contains primops for getting and setting state variables. The final L1 fragment that is generated for the (b→SF b c) component of switch# (post partial evaluation phase) is given below:

```c
... -- s1 and s2 recursively in scope
let x = readInt() in
((λ val . if(val % 2 == 0) then (λ val2 . let temp = s1 in let _ = (set s1 to val2) in writeInt (f temp)) else (λ val3 . let temp = s2 in let _ = (set s2 to val3) in writeInt (f temp))) x) x ....
```

From the above L1 fragment, the C like imperative code shown in Fig. 11 is generated. The if-then-else expressions are translated to case expressions in Erlang and the LLVM translation is very similar to the C code. The global state map now stores two state variables which are updated depending on the value of x. To make the operational aspect of the combinator clearer, we show the evolution of the state variables through the various timesteps of the program:

INPUT: 2 3 4 5 6 7 8...

s1: 0 2 2 4 4 6 6...
s2: 1 1 3 3 5 5 7...

OUTPUT: 0 3 4 9 8 15 12...

4.4 Pull Semantics

In this section we discuss the semantics of data consumption in our compiler. There exist two principle approaches (1) demand-driven pull and (2) data-driven push of data. As Hailstorm’s signal function semantics assume a continuous streaming flow of data triggering the dataflow graph, our compiler adopts a pull based approach which continuously polls the I/O drivers for data. The program blocks until more data is available from the I/O drivers.

Let us take the earthquake detection example from Section 2.4. The getSample input function in the dataflow graph (Fig. 4) is a
wrapper around the driver for a simulated seismometer, which
when polled for data provides a reading. The rightmost edge of
the graph for the Detection Event pulls on the dataflow graph after
it completes an action and the rest of the graph in turn pulls on
getSample, which polls the simulated seismometer.

However, in certain devices such as UART, a push based model
is more prevalent, where data is asynchronously pushed to the
drivers. In such cases, to avoid dropping data the wrapper function
(such as getSample) needs to be stateful and introduce buffers that
store the data. The pseudocode of the wrapper function for such a
driver, written for our Erlang backend, would look like Fig 12.

```erlang
loop(State)
  receive
    (hailstormcall, From, Datasize) ->
      {Ok, Data} = extract_Data(Datasize, State),
      loop(NewState);
    (uartdriver, Message) ->
      NewState = buffer(Message, State),
      loop(NewState);
  end.
end.
```

Figure 12: Enforcing pull semantics on push-based data

In Fig 12 there are two separate message calls handled. The data
transmission from the drivers is handled using the uartdriver
message call, which continuously buffers the data. On the other
hand, the Hailstorm program, upon finishing one cycle of compu-
tation, requests more data using the hailstormcall message, and
proceeds with the rest of the cycle.

4.4.1 Limitation. Elliott has criticized the use of pull semantics
[19] as being wasteful in terms of the re-computation required in the
dataflow graph. He advocates a hybrid push-pull approach, which,
in case of continuously changing data, adopts the pull model, but in
the absence of any change in the data doesn’t trigger the dataflow
graph. This approach could be useful in resource constrained de-
vice, where energy consumption is an important parameter, and
we hope to experiment with this approach in future work.

4.5 The digital - analog interface

The runtime of Hailstorm has to deal with the boundary of discrete
digital systems and continuous analog devices. The input drivers
have to frequently discretize events that occur at some unknown
point of time into a stream of discrete data. An example is the
Stopwatch simulation from Section 2.5.1. The pressing of an ON
button in a stopwatch translates to the stream of ones (111...) and
when switched OFF the simulation treats that as a stream of zeroes.
This stream transformation is handled by the wrapper functions
around the input drivers.

On the contrary, for the output drivers a reverse translation
of discrete to continuous is necessary. We can take the example
of operating traffic lights (related demonstration in Section 5.1.2).
When operating any particular signal like GREEN supplying a
discrete stream of data (even at the lowest granularity) will lead to
a flickering quality of the light. In that case the wrapper function
for the light drivers employs a stateful edge detector, as discussed
in Section 2.4, to supply a new signal only in case of change.

4.6 Backend specific implementation

The backend implementation includes

- **Memory management.** The compiler attempts to minimize
  the amount of dynamically allocated memory using lambda-
lifting and inlining such that the respective garbage collectors
  have to work less. Future work hopes to experiment with static
  memory management schemes like regions [61].
- **Tail call optimization (TCO).** Erlang itself does TCO and
  LLVM supports TCO when using the fastcc calling convention.

5 EVALUATION

5.1 Case Studies

In this section we demonstrate examples from the synchronous

5.1.1 Watchdog process. A watchdog process monitors a sequen-
tial order processing system. It raises an alarm if processing an
order takes more than a threshold time. It has two input signals - (1)
order : $SF O \cup \mathbb{B}$ which emits True when an order is placed
and False otherwise, (2) done : $SF D \cup \mathbb{B}$ which also emits True
only when an order is done. For output we use - alarm : $SF A \mathbb{B}$
where an alarm is rung only when True is supplied. In the program
below we keep a threshold time for order processing as 3 seconds.

```erlang
def f ((order : $\mathbb{B}$, done : $\mathbb{B}$),
  (time : Int, openOrder : $\mathbb{B}$)) : ($\mathbb{B}$, (Int, $\mathbb{B}$))
  = if
    (openOrder == True & & time > 3)
    then
      (True, (time + 1, False)) -- set alarm once
    else if
      (done == True)
      then
        (False, (0, False)) -- reset
      else if (order == True)
        then
          (False, (0, True))
        else
          (False, (time + 1, openOrder)))

def watchdog : $SF (\emptyset \cup D \cup A) \cup \mathbb{B}$ = (order &&& done) >>
  (loop# f) >> alarm

def main : $SF (\emptyset \cup D \cup A) \cup \mathbb{B}$ = rate# 1.0 watchdog
```

5.1.2 A simplified traffic light system. We take the classic example
of a simplified traffic light system from the Lustre literature [53].
The system consists of two traffic lights, governing a junction of
two (one-way) streets. In the default case, traffic light 1 is green,
traffic light 2 is red. When a car is detected at traffic light 2, the
system switches traffic light 1 to red, light 2 to green, waits for 20
seconds, and then switches back to the default situation.

We use a sensor - sensor : $SF S \cup \mathbb{B}$ - which keeps returning
True as long as it detects a car. The system, upon detecting a sub-
sequent car, resets the wait time to another 20 seconds. We sample
from the sensor every second. For the traffic lights, we use 1 to
indicate green and 0 for red.
A railway level crossing

The problem. We consider a two-track railway level crossing area that is protected by barriers, that must be closed in time on the arrival of a train, on either track. They remain closed until all trains have left the area. The barriers must be closed 30 seconds before the expected time of arrival of a train. When the area becomes free, barriers could be opened, but it’s not secure to open them for less than 15 s. So the controller must be warned 45 s before a train arrives. Since the speed of trains may be very different, this speed has to be measured, by detecting the train at two points separated by a known distance. A first detector is placed 2500 m before the crossing, and a second one 100 m after this first. A third is placed by a known distance. A first detector is placed 2500 m before the crossing area, and a second one 100 m after this first. A third is placed after the crossing area, and records a train’s leaving. We divide our solution into three programs.

The Detect Process. The passage of a train is detected by a mechanical device, producing a True pulse - pulse : SF P () Bool - only when a wheel runs on it (otherwise False). The detect process receives all these pulses, but warns the controller only once, on the first wheel. All following pulses are ignored.

A Track Controller. On each track, a controller receives signals from two detectors. From Detect1 and Detect2, it must compute the train’s speed, and warn the barriers 45 seconds before expected time of arrival at crossing. At the maximum speed of 180 km/h, the 100 m between Detect1 and Detect2 are covered in 2 s. So, a clock pulse every 0.1 s would be of good accuracy.

When a train approaches, the trackController calculates the time for the train to reach the barrier and sends that value. In the absence of a train it sends zeroes.

The Barriers Controller. It consists of an alarm - alarm : SF P1 ∪ P2 () Bool which consumes the time values from the trackController and returns True when an alarm is to be rung. The trackController is also sampled every 0.1 second.

Finally, before sending the signal to the actuators (i.e. the barriers), we need an additional system clock that keeps each barrier open for 45 seconds, and ignores other signals in the interim. The handling of the conversion of discrete signals to continuous is done by the drivers for the actuators, as discussed in Section 4.5.
The memory spike visible upon termination is the garbage collector processing. We use the erlang:statistics module for measuring the actual program and its associated bookkeeping structures takes up an average of 1.5 MB of memory in the programs above. The bytecode-interpreter which occupies 30 MB of memory on average.

Each of the numbers are an average of twenty iterations of data. The above values are all measured at the same interval of 0.1 seconds. Hailstorm currently doesn’t have well defined semantics for programs with multiple clock rates.

### 5.2 Microbenchmarks

Here we provide memory consumption and response-time microbenchmarks for the examples presented above using the Erlang backend.

![Figure 13: Memory consumption of programs](image)

We measure the mean memory consumption for each program over five runs, each of five minutes duration. Given the I/O driven nature of the programs, the memory consumption shows little to no fluctuations. The Erlang runtime (ERTS) upon initialization sets up the garbage collector, initializes the lookup table and sets up the bytecode-interpreter which occupies 30 MB of memory on average. The actual program and its associated bookkeeping structures takes up an average of 1.5 MB of memory in the programs above. The garbage collector remains inactive throughout the program run. The memory spike visible upon termination is the garbage collector pausing the program and collecting all residual memory.

<table>
<thead>
<tr>
<th>Program</th>
<th>Run1 (ms)</th>
<th>Run2 (ms)</th>
<th>Run3 (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>watchdog</td>
<td>7.7</td>
<td>8.65</td>
<td>11.4</td>
</tr>
<tr>
<td>traffic-light</td>
<td>3.81</td>
<td>3.04</td>
<td>2.12</td>
</tr>
<tr>
<td>train controller</td>
<td>29.72</td>
<td>28.05</td>
<td>29.8</td>
</tr>
</tbody>
</table>

Table 1: Response time measured in milliseconds

Table 1 shows the response time for the programs measured in milliseconds. We measure the CPU Kernel time (CPUT) - which calculates the time taken by the dataflow graph to finish one cycle of computation. We show three separate runs where the Erlang virtual machine is killed and restarted to reset the garbage collector. Each of the numbers are an average of twenty iterations of data processing. We use the erlang:statistics module for measuring time and in the applications I/O happens over the command line interface, which explains the overall slow behaviour (tens of milliseconds). The metrics are run on the erats-10.6.4 runtime and virtual machine running on a Macbook-Pro with a 2.9 GHz Intel Core i9 processor. A common observation is that the computation takes less than 1% of the total wall clock time involved in the response rate, showing that the I/O reading/writing times dominate the final response rate.

An alternate benchmarking strategy which we used was to model the input from the sensors as an in-memory structure (in Erlang) and compute the total response time for processing those values using the timer:tc module:

<table>
<thead>
<tr>
<th>Program</th>
<th>Run1 (µs)</th>
<th>Run2 (µs)</th>
<th>Run3 (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>watchdog</td>
<td>97.3</td>
<td>106.7</td>
<td>98.7</td>
</tr>
<tr>
<td>traffic-light</td>
<td>110.8</td>
<td>120.2</td>
<td>115.1</td>
</tr>
<tr>
<td>train controller</td>
<td>144.3</td>
<td>128.1</td>
<td>138.7</td>
</tr>
</tbody>
</table>

The above values are all measured in microseconds which are averaged over forty iterations each. As expected, using an in-memory structure results in graph processing times that are much lower than those in Table 1.

### 6 RELATED WORK

#### 6.1 Programming Languages for IoT

There has been recent work on designing embedded DSLs (EDSLs) for IoT applications [11]. In EDSLs, I/O is handled by embedding a pure core language inside a host language’s I/O model - which is an approach that Hailstorm deliberately avoids. Given the I/O dominated nature of IoT apps, we choose to focus much of our attention on designing a composable stream based I/O model, rather than only considering the pure core language, as many EDSLs do.

Other approaches like Velox VM [62] runs general purpose languages like Scheme on specialized virtual machines for IoT devices. A separate line of work has been exploring restrictive, Turing-incomplete, rule-based languages like IoTDSL [2] and CyprIOT [9].

Juniper [29] is one of the few dedicated languages for IoT but it exclusively targets Arduino boards. Emfrp [56] and its successor XFRP [59] are most closely related to the goals of Hailstorm. However, their model of I/O involves writing glue code in C/C++ and embedding the pure functional language inside it. Hailstorm has a more sophisticated I/O integration in the language.

While IoT stands for an umbrella term for a large collection of software areas, there has been research on declarative languages for older and specialized application areas like:

- **Wireless Sensor Networks (WSNs).** There exists EDSLs like Flask [42] and macroprogramming languages like Regiment [45] and Kairos [24] for WSNs.
- **Real Time Systems.** Synchronous language like Esterel [10], Lustre [26] are a restrictive set of languages designed specifically for real time systems. Further extensions of these languages like Lucid Synchronce [12], ReactiveML [44], Lucy-n [43] have attempted to makes them more expressive.

The applications demonstrated in the paper are expressible in synchronous languages, albeit using a very different interface from Hailstorm. While languages like Lustre and its extensions are pure...
they restrict their synchronous calculus to the pure core language and handle I/O using the old stream based I/O model of Haskell [48]. Hailstorm explores the design space of pure functional programming with the programming model and purity encompassing the I/O parts as well.

In Lustre, a type system called the clock calculus ensures that programs can run without any implicit buffering inside the program. Strong safety properties such as determinism and absence of deadlock are ensured at compile time, and programs are compiled into statically scheduled executable code. This comes at the price of reduced flexibility compared to synchronous dataflow-like systems, particularly in the ease with which bounded buffers can be introduced and used. Mandel et al have studied n-synchronous systems in an attempt to bring greater flexibility to synchronous languages [43]. Hailstorm, in the presence of recursion, is unable to statically predict memory usage but we plan future work on type level encoding of buffer sizes to make memory usage more predictable. Polychronous languages like SIGNAL [8] and FRP libraries like Rhine [6] provide static guarantees on correctness of systems with multiple clocks - something that we hope to experiment with in the future.

6.2 FRP

Hailstorm draws influence from the FRP programming model. Since the original FRP paper [18], it has seen extensive research over various formulations like arrowized FRP [46], asynchronous FRP [16], higher-order FRP [35], monadic stream functions [47].

Various implementations have explored the choice between a static structure of the dataflow graph (for example Elm [15]) or dynamic structure, as in most Haskell FRP libraries [3] as well as FrTime in Racket [13]. The dynamic graph structure makes the language/library more expressive, allowing programs like sieves [25].

The higher-order FRP implementation offers almost the local maxima of tradeoffs, but at the cost of an extremely sophisticated type system, which infests into the source language, compromising its simplicity.

The \texttt{loop} combinator in Hailstorm is similar to the \texttt{μ}-combinator first introduced by Sheeran [57] and to the \texttt{loop} combinator in the Causal Commutative Arrows (CCA) paper [40]. CCA presents a number of mathematical laws on arrows and utilizes them to compile away intermediate structures and generate efficient FRP code - a promising avenue for future work in Hailstorm.

FRP has seen adoption in various application areas. Application areas related to our target areas of IoT applications include robotics [64], real-time systems [63], vehicle simulations [21] and DSLs discussed in detail in the previous section.

6.3 Functional I/O

A detailed summary of approaches to functional I/O was presented by Gordon et al. [22]. Since then monadic I/O [23] has become the standard norm for I/O in pure functional languages, with the exception of Clean’s I/O system [1] based on uniqueness types.

More recently, there have been attempts at non monadic I/O using a state passing trick in the Universe framework [20]. The latest work has been on the notion of resource types, proposed by Winograd-Cort et al. [66], which is explored as a library in Haskell. In effect, their library uses Haskell’s monadic I/O model; however, they mention possible future work on designing a dedicated language for resource types. Hailstorm explores that possibility by integrating the idea of resource types natively in a language’s I/O model. FRPNow! [49] provides an alternate monadic approach to integrate I/O with FRP.

7 FUTURE WORK

Hailstorm is an ongoing work to run a pure, statically-typed, functional programming language on memory constrained edge devices. As such, there are a number of open avenues for research:

- **Security.** We hope to integrate support for Information Flow Control (IFC) [28] - which uses language based privacy policies to determine safe dataflow - in Hailstorm. Given that the effectful combinators are based on the Arrow framework, we expect to build on two lines of work - (1) integrating IFC with the Arrow typeclass in Haskell [37] (2) a typeclass based technique to integrate IFC without modifying the compiler [52] but one which relies on the purity and static typing of the language.

- **Reliability.** Hailstorm currently doesn’t provide any fault tolerance strategies to mitigate various node/communication failures. However, being hosted on the Erlang backend, we plan to experiment with distributed versions of the arrow combinators where the underlying runtime would use supervision trees to handle failures. This would be a much more invasive change as the synchronous dataflow model of the language is not practically suitable in a distributed scenario - leading to more interesting research directions on macro-programming models [24].

- **Memory constrained devices.** The evaluation of this paper is carried out on GRiSP boards which are relatively powerful boards. We are currently working on developing a small virtual machine which can interpret a functional bytecode instruction set and run on much more memory constrained devices like STM32 microcontrollers.

8 CONCLUSION

We have presented the design and implementation of Hailstorm, a domain specific language targeting IoT applications. Our evaluation suggests that Hailstorm can be used to declaratively program moderately complex applications in a concise and safe manner. In the future, we hope to use the purity and type system of Hailstorm to enforce language-based security constraints, as well as to increase its expressiveness to enable the description of interacting IoT devices in a distributed system.

ACKNOWLEDGMENTS

This work was funded by the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023). We would also like to thank Henrik Nilsson and Joel Svensson for their valuable feedback on improving our paper.

REFERENCES


