Functional Reactive EDSL with Asynchronous Execution for Resource-Constrained Embedded Systems*

Sheng Wang† Takuo Watanabe‡
Department of Computer Science, Tokyo Institute of Technology

June 1, 2019

Abstract

This paper presents a functional reactive embedded domain-specific language (EDSL) for resource-constrained embedded systems and its efficient execution method. In the language, time-varying values changes at discrete points of time rather than continuously. Combined with a mechanism to let users designate the update interval of values, it is possible to derive the minimal value-updates required to produce the user-desired output. Also, the event-driven backend asynchronously updates an input value when its value is required. In this way, we can greatly reduce the number of updates.

Keywords: Functional Reactive Programming; Embedded Domain-Specific Language; Embedded Systems; Haskell

1 Introduction

A *reactive system* responds to external inputs in a timely manner. Robots that produce real-time motor commands according to continuously-changing environments and GUI systems whose contents change asynchronously with user inputs are two typical examples of reactive systems. In traditional sequential programming languages, we frequently use polling and/or event-driven callbacks to implement such systems. Unfortunately, these methods are complex and error-prone [3]. From the point of view of modularity, polling loops are not composable, and callbacks make control logic scattered across multiple pieces. What's worse, since the arrival of input events is unpredictable, programmers need to manage updates of mutable states carefully to preserve dependencies among variables.

Functional Reactive Programming (FRP) is a programming paradigm originated from dataflow languages. FRP introduces *time-varying values* (aka *signals*) to represent values that (continuously or discretely) change over time. Time-varying values can be composed and transformed as if they were plain values. By applying functional

*To appear in *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, Studies in Computational Intelligence, Springer, 2019.

†kikyouer@gmail.com
‡takuo@acm.org

primitives such as `map`, `reduce` and `filter` on time-varying values, programmers can model the time-dependent relationships declaratively.

While FRP has been successfully applied in many areas such as computer animation and music composing, its usage on embedded systems is still limited. CFRP [14], Emfrp [13] and Juniper [9] tried to fill the gap by generating code with a small memory footprint. However, inefficiency still persists. These languages repetitively sample every input value and propagates update iteratively even if some input values do not contribute to the computation according to data dependencies. The unnecessary frequent activation of input sensors will significantly increase the battery consumption of the device.

The objective of our research is to propose an efficient FRP language for resource-constrained embedded systems. The runtime should be aware of unnecessary updates and automatically removes them from update iterations. We also aim to provide a language that is familiar to Haskell users and is easily extendable.

We present Hae, a code-generating embedded domain-specific language (EDSL) in Haskell. Instead of developing a standalone DSL, we use the technique called *deep embedding* [8]. A deeply embedded DSL overloads the host language's constructs and use them as combinators to construct the abstract syntax. Hae's user programs are written in Haskell source files and preprocessed by Haskell compiler. This process allows users to not only reuse all developing tools of Haskell but also utilize Haskell as a macro system for metaprogramming.

The FRP construct of Hae is different from that of languages such as CFRP or Emfrp. Time-varying values changes at discrete points of time rather than continuously. Combined with a mechanism to let users designate the update interval of values, we make it possible to derive the minimal value-updates required to produce the user-desired output. A Hae program is transformed to C++ code to be integrated with Hae's event-driven backend that asynchronously updates input values when they are required. In this way, we can greatly reduce the number of updates.

The rest of the paper is organized as follows. Section 2 briefly describes Hae using an example. Then, the execution model of the language with its optimization is discussed in Section 3. Section 4 describes the implementation of the language and Section 5 presents the evaluation result. Section 6 overviews related work and Section 7 concludes the paper.

2 Language Hae

2.1 Overview

Hae[1] is an embedded domain-specific language (EDSL) in Haskell that generate C++ code to be integrated with different embedded developing tools. Hae provides users with essential FRP primitives and combinators to compose a static *signal graph*. A signal graph is a directed acyclic graph whose nodes and edges are time-varying values and their dependencies. Hae compiler translates the signal graph to C++ representation, which users can link against different event-driven backends such as Mbed OS[2]. Hae also provides meta-level libraries such as a static Vector library upon basic language elements. The data structure provided by these libraries exist only at compile-time.

[1] `https://github.com/psg-titech/hae`
[2] `https://www.mbed.com/en/platform/mbed-os/`

```
import Hae.Expr
import Hae.Num
import Hae.Bool
import Hae.Compiler

tmp :: E Double -- temperature sensor input
tmp = input "tmp" (EveryS 1)

hum :: E Double -- humidity sensor input
hum = input "hum" (EveryS 1)

di :: E Double -- discomfort index
di = 0.81 * tmp + 0.01 * hum * (0.99 * tmp - 14.3) + 46.3

fan :: E Bool
fan = di .>= 75.0

main = putStrLn $ compile [OutputDef "fan" fan]
```

Listing 1: A simple fan controller in Hae

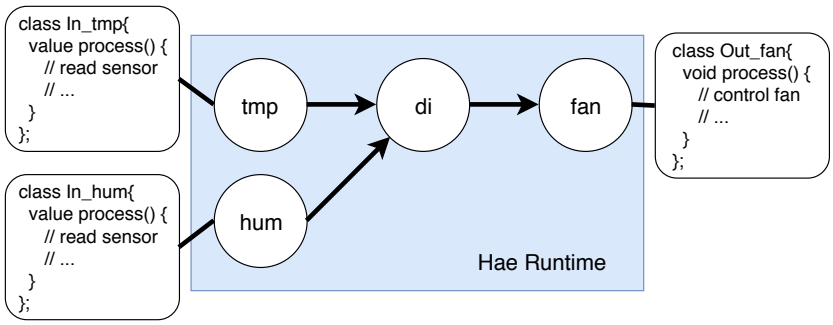

Figure 1: Fan controller after compilation

A simple implementation of a fan controller in Hae is shown in Listing 1. We declare two input signals `tmp` and `hum` of type `Double` to represent the readings of temperature and humidity sensors respectively. The discomfort index[3] is computed from the current temperature and humidity. If the value is larger than 75.0, the value of signal `fan` will be `True`. This means that the fan is turned on while the air environment is uncomfortable. Users need to fill in I/O code in C++ for input (`tmp` and `hum`) and output (`fan`) nodes. After compilation, signals will be transformed to nodes in Hae's C++ runtime, as depicted in Figure 1. As we can see, Hae is a purely functional language where side effects can only happen within I/O nodes.

2.2 Deep Embedding

We claim that the code-generating EDSL in Haskell fits with FRP on resource-constrained systems very well in the following viewpoints:

[3] A kind of human stress indicators. It is empirically known that 50% of people feel uncomfortable if it reaches 75.

- It enables us to reuse Haskell's modules system, type-checker, debugger and other tools, enjoying not only the ease of implementation but also a familiar development environment.
- We can implement functional language features such as higher-order functions and currying effortlessly. The host language will expand usages of these features at Hae's compile-time so that there is no runtime penalty.
- By embedding our language in Haskell, we automatically obtain a powerful and hygienic macro system. It is extremely useful for resource-constrained systems. For example, the `Vector` library built upon the macro system is able to fuse producer and consumer functions and eliminate all intermediate results. Such libraries are also easy to use as we can distinguish them from plain functions by their type signatures. This is further discussed in Section 2.5.

2.3 Discrete Signals

FRP with everchanging continuous signals is easy to understand and elegant: programmers assume that the system runs infinitely fast, and the precision depends merely on iteration interval chosen by the runtime. This may be the best for computer animation that FRP is originally designed for. But when it comes to resource-constrained hardware, we have to take efficiency into serious consideration. Iteration that is too fast causes high power consumption, and slow iteration makes the system unresponsive. As the continuous semantic forces the whole system to iterate at the same speed, it is difficult to choose a proper rate.

Event-driven discrete signals allow finer control over the updating process. Different signals can update asynchronously. We leveraged this feature of discrete FRP and designed a mechanism to perform updates only when they are needed. The mechanism is one of the main contributions of our work and is discussed in detail in Section 3.3.

Another advantage of discrete signals is that it is easier to incorporate asynchronous computation. If some computation is slow and we do not need the latest result of it, we can disconnect it from the main signal graph and use asynchronous signals to connect the input and output of that computation back to the system.

2.4 Signal Definitions

Embedded in Haskell, Hae shares the basic syntax and type system with its host language. As a result, Hae inherits the type definition, function definition and expressions like let-binding directly from Haskell. In this and next subsections, we will introduce how signals and other reactive primitives fit with Haskell.

We use type constructor `E` to denote a signal. The code snippet in Listing 2 shows definitions of a constant signal, an input signal, and a Boolean signal whose value is obtained by basic arithmetic operations.

As we can see, all literals are automatically lifted to constant signals. Currently, Hae has four primitive types: `Int`, `Float`, `Double`, and `Bool`. They can be compiled to their C++ counterparts. Higher-order signal (signal of signals) is not allowed for the sake of static construction of signal graphs.

The definition of input signal `x` is straightforward using function `input`. The first string parameter is used for Hae compiler to generate a function stub of the same name.

```
a :: E Double
a = 0.5

x :: E Int
x = input "x" (EveryMs 10)

y :: E Bool
y = (3 * x + 1) .> 5
```

Listing 2: Example definitions of signals

```
compile :: [OutputDef] -> String

-- to print out the translated string
main = putStrLn $ compile [OutputDef "out1" out1, ...]
```

Listing 3: Designating output signals

Finally, to complete a Hae program, we need to tell the compiler which output nodes we want to compile. As shown in Listing 3, function `compile` translates a list of output nodes to C++ code as a string.

2.5 Functions

Functions in Hae are categorized into two types. One has type $\mathtt{E}\ \alpha_1 \to \mathtt{E}\ \alpha_2 \to \cdots \to \mathtt{E}\ \alpha_m$ and the other has type $\mathtt{E}\ (\alpha_1 \to \alpha_2 \to \cdots \to \alpha_n)$ where types α_i do not contain type constructor `E`. The first type is essentially a Haskell function that any application of it will be inline-expanded. It also supports currying and higher-order functions. Besides inlined functions, it can also be used as a metaprogramming tool.

The second type of function actually exists at runtime as a node in the signal graph. It is better to use this kind of function if it will be called many times to reduce memory usage. The code in Listing 4 shows examples of usage and conversion from the first to the second type of functions.

```
computeDi :: E Double -> E Double -> E Double
computeDi t h = t - 0.55 * (1 - 0.01 * h) * (t - 14.5)

di = computeDi tmp hum

curriedDi :: E Double -> E Double
curriedDi = computeDi tmp

computeDi' :: E (Double -> Double -> Double)
computeDi' = lift2 computeDi

-- operator |$| is required to apply a lifted function
di' = computeDi' |$| (tmp, hum)
```

Listing 4: Two types of functions

```
data Vector a = Indexed { length :: Int, index :: Int -> a }
type EVector a = Vector (E a)

map :: (a -> b) -> Vector a -> Vector b
map f (Indexed l ixf) = Indexed l (f . ixf)

take :: Int -> Vector a -> Vector a
take n (Indexed l ixf) = Indexed (min n l) ixf

drop :: Int -> Vector a -> Vector a
drop n (Indexed l ixf) =
    Indexed (max 0 (l - n)) (\x -> ixf (x + n))

(...) :: Int -> Int -> EVector Int
(...) m n = Indexed (n - m + 1) (+ m)
```

Listing 5: Definition of `Vector` library

The second type of function can be obtained by *lifting* a function of the first type. Users of existing FRP languages may have noticed that the type signature of our `lift` function is different from that of other languages. For example, `lift2` in the code snippet has type

$$\texttt{lift2} :: (E\ a \to E\ b \to E\ c) \to E(a \to b \to c)$$

while its counterpart in CFRP would be typed as

$$\texttt{lift2} :: (\ a \to\ b \to\ c) \to (E\ a \to E\ b \to E\ c).$$

The difference comes from the fact that Hae is an embedded language. Primitive data types (a, b and c) and functions (in the form of $a \to b$) of the host language cannot be directly reified without wrapping. Having to write `E` everywhere may seem tiresome, but it also gives a clear distinction between elements in Haskell and our DSL.

2.6 Datatype and the Vector Library

One limitation of deeply embedded DSL is that users cannot customize datatypes of the DSL without modifying the interpreter (compiler). A well-known technique to deal with this issue is to construct a small core language consisting of essential datatypes and build meta-level libraries of datatypes upon it using the host language Haskell. The design of Hae language takes the same approach. Here we briefly introduce how such a library can be built to show the expressiveness of Hae language.

Hae comes with a simplified version of Vector library in Feldspar [2]. A vector is like an indexed array except that it does not exist in memory at runtime. The definition of the vector type is shown in Listing 5. Data constructor `Vector` takes the length and the index function (function mapping an index to corresponding value stored in the vector) as parameters. The `map` function simply composes the provided function with the vector's index function. Other functions work similarly.

As we have discussed in Section 2.5, Haskell functions will be automatically inline-expanded. The producer function and consumer function of a vector will be fused together so that there will be no vectors of intermediate results.

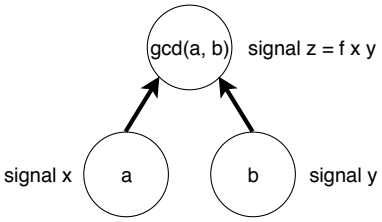

Figure 2: Dependency

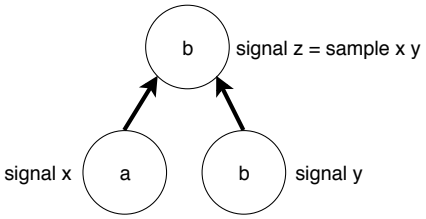

Figure 3: Sampling

Vector provides an efficient implementation of static-sized array in Hae. Other static container types can be easily built using the same method.

3 Execution Model

3.1 Updates on Signal Graph

The execution model of Hae is similar to that of Emfrp [13]. Signals and their dependencies form a directed acyclic graph. There are two ways to propagate updates on such a graph. They can be "pushed" from input nodes or "pulled" by output nodes. Pull-based FRP computes backward through the signal graph whenever results are demanded. As it is demand-driven, it eliminates unnecessary computation or polling on inputs. However, pull-based implementation requires lazy evaluation which limits its usage on resource-constrained systems.

Hae uses a push-based runtime. Comparing to pull-based FRP, it causes less latency between occurrence of an event and the reaction to it [6]. Updates are pushed following the order which is given by the topology sorting of the graph, from source nodes representing input signals to sink nodes representing output signals.

An important difference from Emfrp is that all input signals are not required to be updated in the same update iteration. In Hae, every signal contains information about its updating interval. Asynchronously, some signals update at a faster rate than others. In the iteration where fast signals update, slow signals do not need to activate. The runtime will push their previous value to its descendants. As a result, we no longer need to update the whole system whenever we get a new input.

We give users the ability to explicitly control the updating interval. When declaring input signals, users can specify update intervals of them. The update intervals of other signals that depend on the input signals can be calculated from them. For example, if signal `x` updates every 150ms and signal `y` updates every 200ms, then any signals that depend on these two signals will update every 50ms (calculated by taking the GCD of the two), as shown in Figure 2. We can also construct a new signal by sampling on another signals when its value changes. For example, in the case of `x` and `y`, the updating interval of `sample x y` is 200ms, as shown in Figure 3.

3.2 Stateful Computation

Hae provides two kinds of stateful computation. The first one is to refer to the previous value of a signal. This is implemented by inserting a `delay` node into the signal graph. The node will delay one event occurrence of a signal. If a signal emits

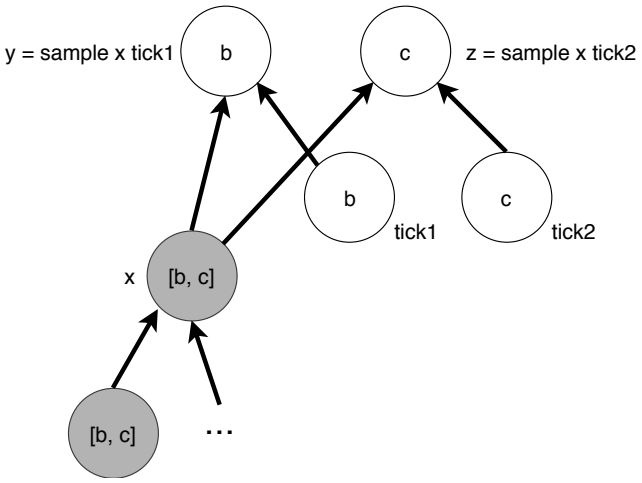

Figure 4: Calculation of actual update interval

`[0,1,2,3...]`, the delayed signal will yield `[0,0,1,2...]` in the corresponding update iteration.

The `@last` modifier in Emfrp serves a similar purpose of referring to previous values. But their semantics are different. Emfrp adopts the continuous signal model, `@last` in Emfrp refers to the value at the snapshot taken a moment ago (implemented as values in the last update cycle). Thus `a@last` and `b@last` are guaranteed to represent values taken in the same update cycle. In contrast, `delay a` and `delay b` in Hae do not necessarily represent values of the same time as updates are asynchronous.

Another kind of stateful computation is `foldp`. It behaves like the fold function in functional languages, but instead of accumulating list elements, it accumulates the history of values of a signal. `foldp` should be used with care since it cannot be optimized using the algorithm below.

3.3 Optimizing Push Timings

It is obvious that when we sample a fast-updating signal on a slower signal, all preceding signals of the sampled signal only need to update at the slower rate. Using this idea, we can calculate the actual update interval (which should be slower) by working from output signals backward the signal graph.

Consider the signal graph in Figure 4. For signal `y` and `z` to update at interval *b* and *c*, signal `x` that provides value to them should update at both the intervals. Thus the actual update interval of signal `x` should be a list, containing those two update intervals. We apply the same method down the dependency graph and build the list of update intervals until reaching input signals.

Things become more complicated when stateful computations are introduced. If `foldp` is used, every previous value of preceding signals will contribute to the final accumulated value. In this case we cannot optimize the update interval.

For stateful computation consisting of `delay` nodes, we need to offset the update timing. Let us extend the denotation of update intervals to offsetted timing using `interval(offset)`. For example, offsetted timing `60ms(-1ms)` means that the signal's value will be required at 59ms, 119ms,... With this extension, the actual re-

quired timing of a signal `actual(x)` can be computed by the following rules:

- Let `dep(x)` be signals that depend on `x`, `int(x)` be the user-designated update interval of `x`.
- If `x` is an output signal, `actual(x) = [int(x)]`.
- If `x` is a delayed signal, for each signal `s` in `dep(x)`, offset each timing in `actual(s)` by `-int(x)` and concatenate them together. For example, if `int(x) = 1ms`, `dep(x) = [y, z]`, `actual(y) = [15ms, 20ms]` and `actual(z) = [50ms(-1ms)]`, then after offsetting and concatenating, `actual(x) = [15ms(-1ms), 20ms(-1ms), 50ms(-2ms)`.
- If `x` is not delayed, we can just concatenate the timings without offsetting them.

4 Implementation

Hae is a code-generating embedded domain specific language. We will show the details of our prototype implementation in this chapter. The construction of a domain specific language within Haskell is explained in Section 4.1. Implementation of the C++ runtime is shown in Section 4.2. Finally, we introduce Hae's compiler in Section 4.3 with emphasis on a new optimization technique.

4.1 EDSL Frontend

The core expression of Hae revolves around type `OpenExpr`. Type `E a`, which represents a signal of type `a`, is just a synonym of type `forall f. (Ref (OpenExpr f a))`. Before explaining the wrapper type `Ref`, let us first show the definition of `OpenExpr` in Listing 6. Some typical constructors of `OpenExpr` are also introduced below.

`OpenExpr` is the basic building block of *parametric higher-order abstract syntax* [4]. The first type parameter `f` is used to capture the type of shared expressions in `LetRec` bindings using the technique introduced by Oliveira *et al.* [12] This technique ensures the type safety of the `LetRec` binder by statically encoding the types of binded expressions in *typed lists*. The Haskell compiler will then be able to reject illegal `LetRec` expressions.

Type constraint `HType t =>` on the `Inp` and `Lit` constructors ensures that we can only define input signals and constant signals of primitive types in Hae (`Int`, `Double`, `Float` and `Bool`).

Predefined primitive functions are reified by `PrimOp` constructor. By making `OE` an instance of `Num` type class, we can let Haskell implicitly insert corresponding `PrimOp` constructors when users write literal values and arithmetic operations. As a result, programming with signals feels no different than programming with plain values.

For user-defined functions that exist at run time (the second kind of function introduced in Section 2.5), we use `Lam`, `Var` and `App` constructors to reify them. `Lam` records the original unlisted function to be given a unique name when compiled. `Var` serves as a dummy argument to capture argument binding of that function. The application of these functions are represented by using `App`.

Finally, we have individual constructors for FRP primitives `SampleOnChange`, `Delay` and `Foldp` to distinct them from other expressions.

```
type E a = forall f. OE f a
newtype OE f a = OE (Ref (OpenExpr f a))

data OpenExpr (f :: * -> *) t where
  Inp :: HType t => String -> TimingDef -> OpenExpr f t
  Lit :: HType t => t -> OpenExpr f t
  PrimOp :: (Typeable a, Typeable b) =>
            PrimOpId -> (a -> b) -> OpenExpr f (a -> b)
  IfThenElse :: (Typeable t) => OE f Bool -> OE f t -> OE f t -> OpenExpr f
    t

  Var :: f t -> OpenExpr f t
  Lam :: (Typeable a, Typeable b) => (f a -> OE f b) -> OpenExpr f (a -> b)
  App :: (Typeable a) => OE f (a -> b) -> OE f a -> OpenExpr f b
  LetRec
    :: (CList ts, Typeable t)
    => (TList f ts -> TList (OE f) ts)
    -> (TList f ts -> OE f t)
    -> OpenExpr f t

  SampleOnChange :: Typeable a => OE f a -> OE f b -> OpenExpr f a
  Delay :: Typeable t => OE f t -> OpenExpr f t
  Foldp :: (Typeable a, Typeable b) => OE f (a -> b) -> OpenExpr f (a -> b)
```

Listing 6: Definition of core expression `OpenExpr`

```
data Ref a = Ref { refId :: Unique, deref :: a }

instance Eq (Ref a) where
  Ref x _ == Ref y _ = x == y
```

Listing 7: Definition of `Ref`

Careful readers may notice that we use type `OE` rather then `OpenExpr` in the recursive constructors. `OE` wraps `OpenExpr` with `Ref`, a data type that enables observable sharing [5] in a purely functional language.

Observable sharing is essential for an embedded domain specific language. Without observable sharing, our compiler cannot discover links among multiple references of a same value. As a result, for example, in expression `C*D*(D*D+D*E)`, `D` will be computed four times.

The definition of `Ref` is shown in Listing 7. It tags our value with a `Unique` identifier. `Ref` adds one level of indirection to raw values so that we can rediscover the sharing when analyzing these wrapped values by comparing their `refId`.

In Hae, users use smart constructors which wraps `Ref` automatically. For example, the smart constructor of `Delay` is defined as:

```
delay :: OE f a -> OE f a
delay = OE . ref . Delay
```

where `ref` is the function responsible to generate a unique `refId`. For simplicity, Hae implements `ref` function using `unsafePerformIO` in Haskell, as shown in Listing 8.

```
import Data.Unique (newUnique)

ref :: a -> Ref a
ref x = unsafePerformIO $ do
          u <- newUnique
          return (Ref u x)
```

Listing 8: Implementation of `ref`

4.2 Runtime

Hae uses a modified version of CFRP's event-driven runtime [14]. They share the same queue-based iteration logic within an update iteration:

1. During each update iteration, a node maintains the number of events it expects to receive.
2. When beginning an update iteration, input nodes are pushed to the update queue.
3. For each node `n` in the update queue, repeat the process until the queue is empty:
   (a) `n.process()` is called to generate an update event. The actual computation of the node is done here
   (b) The update event is sent to all of its children
   (c) If a children has received enough events, it will be added to the update queue
   (d) Finally, `n` is removed from the update queue

In CFRP, this update iteration is repeatedly executed in an infinite loop. To adapt it to our asynchronous execution model, we enhance input nodes with structure `Timing` (Listing 9). Every input node now contains the information of multiple timings. The runtime engine only initiates an update iteration at these timings. In the update iteration, input nodes which are not the initiators of the update will remain deactivated, sending empty update events to its children. After an update iteration is finished, the runtime engine schedules the next activation of these input nodes.

```
struct Timing {
    ttime period; // 64 bit int, the actual update interval
    std::vector<ttime> offsets;
    Timing(...) {...} // constructor
};
```

Listing 9: The data structure `Timing`

Note that `delay` is also a new type of node in Hae. Its implementation is straightforward. When a `delay` node is activated, it saves its newly received value to member variable `prev_` to be used as next iteration's output.

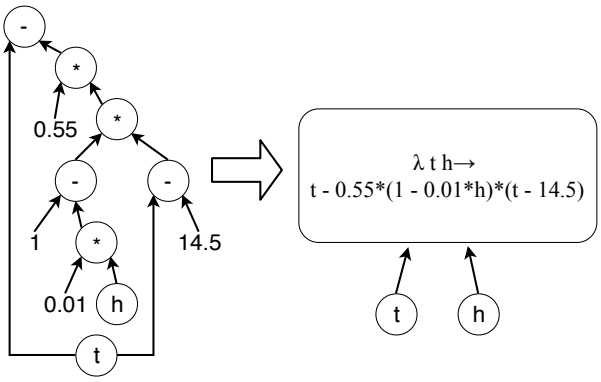

Figure 5: A simple example of node merging

4.3 Compiler and Optimization

The compiling process of an EDSL in Haskell is slightly different from that of a standalone functional language. There is no need for lexing, parsing, α-conversion and even K-normalization as the host language automatically expands let bindings before constructing the DSL's abstract syntax.

We can also let Haskell do type inference for us by using `Data.Typeable` in Haskell's basic libraries. The `Typeable` class associates type representations to types. By deriving `OpenExpr` as an instance of `Typeable`. We can retrieve the type representation of any DSL expression using

$$\texttt{typeOf} :: \texttt{forall}\ a.\texttt{Typeable}\ a \Rightarrow\ a \rightarrow \texttt{TypeRep}.$$

Since Hae generates C++ code, we can let C++ compilers handle most of the optimizations. However, there is still one essential optimization to do: merging adjacent primitive operations into a single node. Without merging, every arithmetic operation becomes an individual node at runtime, which is unacceptable considering time and space overhead.

Figure 5 demonstrates node merging on the computation of comfort index. Nodes of primitive functions and constants are merged into a single node that takes its input from two nodes.

A more general case is shown in Figure 6. A white node denote a primitive or constant node. Other nodes that cannot be merged are marked black. Our node-merging algorithm partitions white nodes into 3 clusters. Each cluster contains exactly one *output node* which is marked gray. The output node must be the sink node within the partition. Finally, nodes in each partition are merged together, forming a simplified signal graph.

The key problem here is how to find the partition. Globally we maintain a map from node to cluster to record the membership of every white node. At the beginning, the map is empty. Then, for each output node, we traverse backward the dependency graph until a white node x that belongs to no cluster is found. This is the gray node of a new cluster. Then we start the subprocess below:

1. Mark node x gray.
2. Use node x as the source node for a new round of DFS. Note that we search only white nodes in this round of DFS.

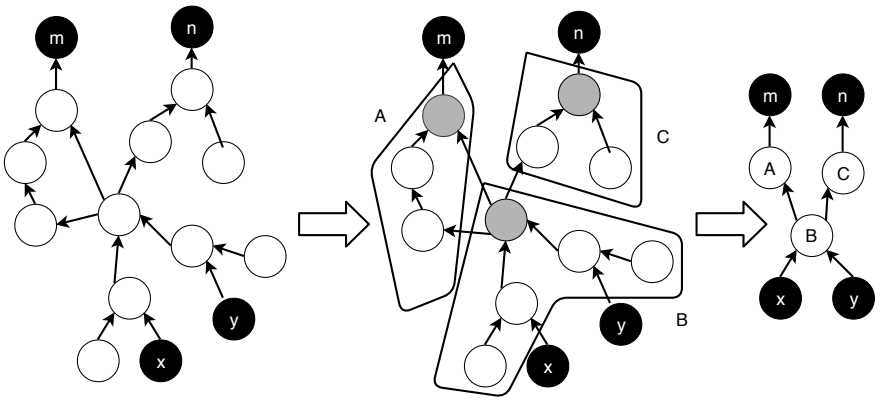

Figure 6: A more general case of node merging

3. During the DFS, set white nodes that previously belongs to no cluster as a member of cluster x. However, if we encountered a white node n that is already a member of another partition, it means cluster x depends on the value of n while n is not an output node. This is not allowed. In this case, we recursively start a new subprocess with $x = n$, effectively making a new partition from n.

The algorithm stops when every white node belongs to a partition.

In Emfrp, the partition of primitive nodes is explicitly defined by the user using keyword `node`. Hae makes this process automatic and optimal. Users can write expressions in whatever way they want without worrying about the overhead of additional intermediate nodes.

5 Evaluation

In this section, we evaluate the primary objective of our research — execution efficiency by two metrics. One is the number of times input sensors are activated. The other is the total number of node updates. Both measured in a fixed period of time. We run a test program using both Hae's asynchronous runtime and traditional repetitive iteration and compare the results of the two metrics. For evaluation purpose, we have built a simple repetitive execution engine upon Hae's asynchronous runtime (Listing 10).

```
void callback_(ttime now) {
  now_ = now;
  iterate_(); // the actual update iteration
  // call me after ITERATE_INTERVAL
  impl_set_cb_(now, ITERATE_INTERVAL);
  impl_yield_();
}
```

Listing 10: Repetitive execution

The test program (Listing 11) is a fan controller that switches on and off every minute depending on the sensor readings of temperature and humidity sensor. We

```
tmp :: E Double
tmp = input "tmp" (EveryMs 100)
hum :: E Double
hum = input "hum" (EveryMs 1000)

avg3 :: E a -> E a
avg3 s = (s + prevS + delay prevS) / 3
  where
    prevS = delay s

tmpAvg = avg3 tmp

computeDi :: E Double -> E Double -> E Double
computeDi t h =
  0.81 * t + 0.01 * h * (0.99 * t - 14.3) + 46.3

di =
  sampleOnChange (compute_di tmpAvg hum) (fps (EveryS 5))

diAvg = avg3 di

fan :: E Bool
fan =
  sampleOnChange diAvg (fps (EveryMin 1)) .>= threshold
  where
    threshold = 75.0

main = putStrLn $ compile [OutputDef "fan" fan]
```

Listing 11: The test program in Hae

set the sampling interval of the temperature sensor `tmp` to 100ms and the humidity sensor `hum` to 1000ms. These values provide a baseline for computation of actual update timings. `avg3` is a helper function that utilizes Hae's discrete-time semantic to compute the average of the latest three values of a signal. We use it to smooth readings of `tmp` sensor. The discomfort index `di` is sampled every 5 seconds using built-in signal generator `fps`. Then, we take the average of discomfort index and use it to determine the switch of the fan once per minute.

Listing 12 shows the two input nodes after compilation using GHC 8.0.2. We can see that `tmp` contains 9 timings and `hum` have 3. During an update iteration, the runtime has to remember which node has initiated this iteration and also when to schedule the next round. Thankfully, the memory usage is linear to the total number of update timings.

We used a Linux-based backend to simulate running the test program for 60 minutes. The code generated by Hae and the backend are compiled by GCC 8.1. We ran the simulation on our PC running Fedora 28 on Intel's Core i7-6700K 4.0GHz with 16 gigabytes of RAM. The backend records the number of times of input node activations and node updates (by counting how many times a node's `process()` method is called). The repetitive reference runtime iterates at the rate of 100ms (the same as the update interval of input signal `tmp`). Results are shown in Table 1. Both metrics are reduced by orders of magnitudes.

```
// ...
In_tmp n33;
n33.add_timing(hae::Timing(60000, 0, 0));
hae::ttime t33_1[8] = {-10200, -10100, -10000,
                       -5200, -5100, -5000, -200, -100};
n33.add_timing(hae::Timing(60000, 8, t33_1));
engine.register_input_node(&n33);
// ...
In_hum n47;
n47.add_timing(hae::Timing(60000, 0, 0));
hae::ttime t47_1[2] = {-10000, -5000};
n47.add_timing(hae::Timing(60000, 2, t47_1));
engine.register_input_node(&n47);
// ...
```

Listing 12: Input nodes after compilation

Table 1: Comparison of efficiency (in number of times)

	Iterations	Input Activations	Node Updates
Repetitive	36000	72000	648000
Async Exec	540	722	4509

For Hae's asynchronous runtime, increasing the final sampling of `fan` from 60s to 120s will result in both the number of times of input activations and node updates in Hae's runtime being in half. Increasing the update interval of `tmp` or `hum` causes no change to the result because the number of timings in the test program is determined by the sample node. In contrast, for the repetitive runtime, making the final sampling slower makes no difference. But increasing the update interval of input signals effectively means we can iterate at a slower rate, resulting in fewer updates. To summary up, compared to traditional languages, Hae's runtime perform better when output signals operate slower and when input sensors run at a faster rate.

6 Related Work

6.1 FRP Languages for Small-scale Systems

Flask [11] is a continuous FRP language targeting sensor networks. The two-stage language design separates the meta-language describing sensor network structure and the node-level language that compiles to NesC, a C-like language for sensor nodes. The runtime deploys node-level code to each sensors and constructs the network. Native NesC code can be embedded in Flask by using Haskell's quasiquotation language extension.

Emfrp [13] achieved static memory footprint as a purely functional reactive language. An Emfrp program can be directly mapped to a static directed graph, eliminating recursion and any dynamic allocation of memory. Signals (called Nodes) in Emfrp are not first-class citizens. To reference a node one must supply with the name of it. The graph-like program is then transformed to C/C++ codes of the target platform with stub I/O functions for input and output nodes. Programmers fill in these stubs to

connect the reactive part to other parts of the system.

Juniper [9] is a ML-like FRP language targeting the Arduino platform. Unlike Emfrp which limits the language's expressive power, Juniper is equipped with advanced language features such as anonymous functions and parametric polymorphism. The signal graph is dynamic in Juniper, allowing use of higher-order signals. Juniper is not a purely functional language.

6.2 Code-Generating Embedded DSL

There are two flavors of domain-specific language. A DSL can be either first-class, with its own compiler or interpreter, or embedded in a host language. The embedded approach has advantages in that it can utilize the host language's syntax and ecosystem.

Haskell is a functional language with powerful type system and overloading capabilities. Previous research has explored Haskell's ability to not only run DSLs in its own runtime, but also generate code that can be interpreted by external programs. Code-generating embedded DSL leverages both the syntactic convenience of the host language and the flexibility of a customized runtime.

Elliott *et al.* demonstrated techniques to capture and reify variable bindings in Haskell in their image-processing language Pan [7]. Value sharing and recursion in EDSLs is made possible by using I/O based *observable sharing* [5] or *parametric higher-order abstract syntax* [4]. Some languages, such as [1, 2, 10], are built upon these techniques to generate all kinds of codes from CUDA, DSP algorithms to even Haskell code itself.

7 Conclusion

We have designed and implemented Hae, a functional reactive programming language targeting small-scale embedded systems. Hae showed that we can rule out unnecessary updates by combining the discrete-time semantic and explicit update intervals. This makes Hae much more efficient than previous FRP languages for embedded devices.

The design choice of a code-generating embedded domain specific language let us take advantage of a familiar developing environment, a powerful macro platform, flexible choice of backends and ease of implementation. The algorithm we used for merging primitive nodes is not only essential for embedded DSLs but also useful for standalone FRP languages.

Hae is currently a prototype language that requires polishing. Some designs may not be the best choice for embedded development. More case studies about FRP and embedded programming need to be done, especially those that may benefit from metaprogramming. The survey will not only inspire new ideas but also provide better ways to evaluate our work.

As for language features, the ability to change update intervals at runtime can be an interesting research direction. While this will require more information about signal nodes to be maintained at runtime, the flexibility it brings should outweigh the cost.

Acknowledgments

This work is supported in part by JSPS KAKENHI Grant No. 18K11236.

References

[1] Ankner, J., Svenningsson, J.: An EDSL approach to high performance Haskell programming. In: ACM SIGPLAN Symposium on Haskell (Haksell 2013), pp. 1–12. ACM (2013). doi:10.1145/2503778.2503789

[2] Axelsson, E., Claessen, K., Sheeran, M., Svenningsson, J., Engdal, D., Persson, A.: The design and implementation of Feldspar: An embedded language for digital signal processing. In: IFL 2010: Implementation and Application of Functional Languages, *Lecture Notes in Computer Science*, vol. 6647, pp. 121–136. Springer (2010). doi:10.1007/978-3-642-24276-2_8

[3] Bainomugisha, E., Carreton, A.L., Van Cutsem, T., Mostinckx, S., De Meuter, W.: A survey on reactive programming. ACM Computing Surveys **45**(4), 52:1–52:34 (2013). doi:10.1145/2501654.2501666

[4] Chlipala, A.: Parametric higher-order abstract syntax for mechanized semantics. In: 13th ACM SIGPLAN International Conference on Functional Programming (ICFP 2008), pp. 143–156. ACM (2008). doi:10.1145/1411204.1411226

[5] Claessen, K., Sands, D.: Observable sharing for functional circuit description. In: Advances in Computing Science (ASIAN '99), *Lecture Notes in Computer Science*, vol. 1742, pp. 62–73. Springer (1999). doi:10.1007/3-540-46674-6_7

[6] Elliott, C.: Push-pull functional reactive programming. In: Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell 2009), pp. 25–36. ACM (2009). doi:10.1145/1596638.1596643

[7] Elliott, C., Finne, S., de Moor, O.: Compiling embedded languages. Journal of Functional Programming **13**(3), 455–481 (2003). doi:10.1017/S0956796802004574

[8] Gill, A.: Domain-specific languages and code synthesis using Haskell. ACM Queue **12**(4) (2014). doi:10.1145/2611429.2617811

[9] Helbling, C., Guyer, S.Z.: Juniper: A functional reactive programming language for the Arduino. In: 4th International Workshop on Functional Art, Music, Modelling, and Design (FARM 2016), pp. 8–16. ACM (2016). doi:10.1145/2975980.2975982

[10] Mainland, G., Morrisett, G.: Nikola: Embedding compiled GPU functions in Haskell. In: 3rd ACM Symposium on Haskell (Haskell 2010), pp. 67–78. ACM (2010). doi:10.1145/1863523.1863533

[11] Mainland, G., Morrisett, G., Welsh, M.: Flask: Staged functional programming for sensor networks. In: 13th ACM SIGPLAN International Conference on Functional Programming (ICFP 2008), pp. 335–346. ACM (2008). doi:10.1145/1411204.1411251

[12] Oliveira, B.C.d.S., Löh, A.: Abstract syntax graphs for domain specific languages. In: Workshop on Partial Evaluation and Program Manipulation (PEPM 2013), pp. 87–96. ACM SIGPLAN, ACM (2013). doi:10.1145/2426890.2426909

[13] Sawada, K., Watanabe, T.: Emfrp: A functional reactive programming language for small-scale embedded systems. In: MODULARITY Companion 2016: Companion Proceedings of the 15th International Conference on Modularity, pp. 36–44. ACM (2016). doi:10.1145/2892664.2892670

[14] Suzuki, K., Nagayama, K., Sawada, K., Watanabe, T.: CFRP: A functional reactive programming language for small-scale embedded systems. In: Theory and Practice of Computation (Proc. WCTP 2016), pp. 1–13. World Scientific (2017). doi:10.1142/9789813234079_0001