Towards Secure IoT Programming in Haskell

Nachiappan Valliappan
Chalmers University of Technology
Gothenburg, Sweden
nacval@chalmers.se

Robert Krook
Chalmers University of Technology
Gothenburg, Sweden
krookr@chalmers.se

Alejandro Russo
Chalmers University of Technology
Gothenburg, Sweden
russo@chalmers.se

Koen Claessen
Chalmers University of Technology
Gothenburg, Sweden
koen@chalmers.se

Abstract

IoT applications are often developed in programming languages with low-level abstractions, where a seemingly innocent mistake might lead to severe security vulnerabilities. Current IoT development tools make it hard to identify these vulnerabilities as they do not provide end-to-end guarantees about how data flows *within and between* appliances. In this work we present Haski, an embedded domain specific language (eDSL) in Haskell for secure programming of IoT devices. Haski enables developers to write Haskell programs that generate C code without falling into many of C's pitfalls. Haski is designed after the synchronous programming language Lustre, and sports a backwards compatible information-flow control extension to restrict how sensitive data is propagated and modified within the application. We present a novel eDSL design which uses recursive monadic bindings and allows a natural use of functions and pattern matching to write embedded programs. To showcase Haski, we implement a simple smart house controller where communication is done via low-energy Bluetooth on the Zephyr IoT OS.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; • **Security and privacy** → *Information flow control.*

Keywords: Synchronous programming, Information-flow Control, eDSL, IoT, Haskell

ACM Reference Format:
Nachiappan Valliappan, Robert Krook, Alejandro Russo, and Koen Claessen. 2020. Towards Secure IoT Programming in Haskell. In *Proceedings of the 13th ACM SIGPLAN International Haskell Symposium (Haskell '20), August 27, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3406088.3409027

Haskell '20, August 27, 2020, Virtual Event, USA

1 Introduction

The Internet of Things (IoT) conceives a future where "things" (embedded electronics) can be interconnected. While a compelling vision, recent events have demonstrated the *high vulnerability* of IoT (e.g., [Bertino and Islam 2017; Fernandes et al. 2016; Schuster et al. 2018; Wang et al. 2018]). Hence, it has become important to develop security solutions which address the concerns of unauthorized access to data and privacy loss.

We believe there are two major aspects which contribute to the current poor state-of-the-art in IoT security: *the chosen programming languages for development* and *the lack of end-to-end guarantees.* IoT development is often done in programming languages (like C) with low-level of abstractions, where a seemingly innocent mistake might lead to severe vulnerabilities like buffer overflows. Similarly, development tools present no end-to-end guarantees about how data flows *within and between* devices—thus making it hard to *confine* sensitive information.

Figure 1 shows the running example throughout this paper: a simplified *smart house controller* called Halexa. Halexa

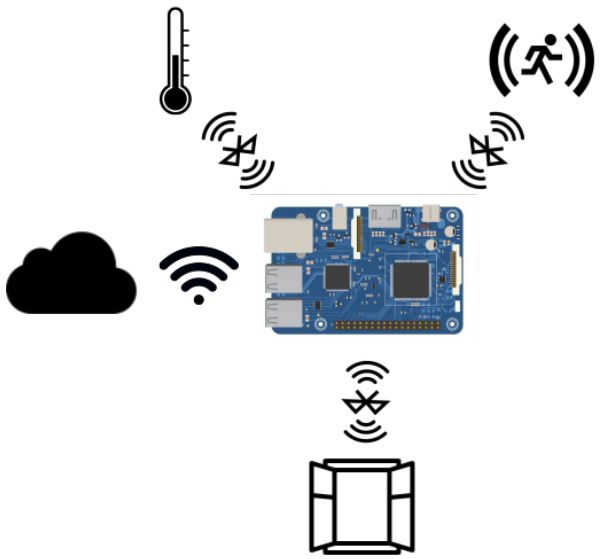

Figure 1. A Smarthouse example

consists of a micro-controller with Wifi access (required to fetch software updates) which is connected to three Bluetooth devices: a thermometer, a motion sensor, and a window. The micro-controller software is responsible for opening the window when it is too hot inside the house. We assume that there is no Air Conditioning in the house—not an uncommon assumption in, for example, Nordic countries. While simple, this scenario presents interesting security and safety concerns: (i) to avoid robbery, windows must only be opened when there is someone at home, and (ii) the motion sensor data should be kept *confined* within the system and not leaked via Internet—leaking it can hint burglars about the vacancy of the house. Observe that the micro-controller needs to have access to the sensors' data in order to deliver its function. Can we use Haskell to program constrained devices and ensure the mentioned security requirements by construction?

In this paper, we present Haski, an embedded domain specific language in Haskell for secure programming of IoT devices. Haski enables developers to write Haskell programs that generate C code without falling into many of C's pitfalls (e.g., those related to memory safety, undefined behavior, etc.). Haski follows the footsteps of the synchronous programming language Lustre [Caspi et al. 1987; Halbwachs et al. 1991], which is an event-driven programming language with strong guarantees on resource usage—a must when programming low-power devices often found in IoT systems. Haski enhances Lustre with confidentiality and integrity security guarantees, as well as a means of communicating with streams generated by callback functions.

By adopting a synchronous programming model, Haski is able to provide resource bounds while removing memory-based security vulnerabilities by construction. Haski's design and implementation is unique compared with previous Haskell eDSLs for Lustre-like languages [Bjesse et al. 1998; Hawkins et al. 2011]. Firstly, Haski presents a novel monadic design which allows programmers to leverage Haskell's monadic bindings (i.e., **do** and **mdo**) to specify streams as literate as possible. Secondly, Haski conceives a new DSL technique to compile Haskell functions on Haski-expressions into callable components of the target language. Finally, Haski provides user-defined enumeration types, where developers can simply use Haskell's **case** expression to inspect them, while raising a type-error in case of non-exhaustive patterns—thus making the code more robust. To address end-to-end guarantees, Haski incorporates information-flow control (IFC) techniques [Sabelfeld and Myers 2003] to restrict how data propagates and gets modified—thus protecting the confidentiality and integrity of data. With IFC, developers can, for instance, incorporate third-party Haski code to analyze sensitive data like that coming from the motion sensor while still preventing data leaks. To keep the types in eDSL simple, Haski enforces IFC at code-generation time by only tracking data propagation among end-points streams indicated by developers, e.g., the thermometer, motion sensor, window and Internet communication channel in Figure 1.

Contributions. The main research contribution of this paper is the design and implementation of Haski. We show how to design a synchronous language that is type-safe, protects confidentiality and integrity of data, handles I/O, and generates C code. Importantly, our design does not require any modifications to GHC or the use of compiler plug-ins. Instead, Haski uses embedding techniques by leveraging advanced type-level features of GHC such as GADTs [Peyton Jones et al. 2006], data kinds [Yorgey et al. 2012], existential types, and pattern synonyms [Pickering et al. 2016]. Some of the techniques developed for Haski can be generalized and used for general DSL design in Haskell.

2 Haski by Example

Haski programs are written in Haskell using a special set of combinators. In this section, we illustrate various examples of Haski programs and showcase these combinators. For the upcoming examples, we use the data type *Action* to represent an action indicating whether our user Octavius has left (or entered) the house.

```
data Action = Left | Entered
```

The purpose of the *Action* data type (instead of, for example, *Bool*) is to illustrate the use of user-defined data types in Haski programs.

Recursive definitions. A Haski program is a collection of stream definitions written in the *Haski* monad. A simple stream can be defined using the *letDef* combinator, which has the following type.

```
letDef :: Stream a → Haski (Stream a)
```

Using Haskell's **do** notation, we can use this combinator to bind streams to variables as follows.

```
left :: Haski (Stream Action)
left = do
  x ← letDef (val Left)
  return x
```

This program defines the constant stream that repeats the action *Left* as $Left, Left, Left, \ldots$ using the combinator $val :: HT\ a \Rightarrow a \rightarrow Stream\ a$. The type constraint *HT* ensures that a type is recognized by the Haski compiler and can be compiled by it. In this case, we may suppose that *Action* already satisfies this constraint, but we will later see how this is made possible.

Streams may also be defined recursively using the *fby* combinator (read *followed by*).

```
fby :: HT a ⇒ a → Stream a → Haski (Stream a)
```

The stream v `` `fby` `` s begins with the value v and is followed by the stream s. For example, we can define a stream of alternating actions such as *Left*, *Entered*, *Left*, *Entered*, … using the *fby* combinator as follows.

```
alt :: Haski (Stream Action)
alt = mdo
  x ← Left `fby` y
  y ← Entered `fby` x
  return x
```

The stream x here defines a stream that begins with *Left* and is followed by y. Similarly, y begins with *Entered* and is followed by x. We use the keyword **mdo**[1] instead of **do** for (mutually) recursive definitions.

Pattern matching definitions. Streams can also be defined by pattern matching on values of other streams using the *match* combinator.

```
match :: (FinEnum a, Streams b) ⇒ Stream a → (a → b) → Haski b
```

The combinator application *match e f* defines the streams resulting from applying the observed value of e to f. The definition of f enables pattern matching on the value of e. The type constraint *FinEnum* subjects the type a to be *finitely enumerable*, and the constraint *Streams* overloads the type b to allow the function f to return multiple streams such as lists or tuples of streams. The constraint *FinEnum* ensures that *match* can only be used to pattern match on streams with finitely many values—a restriction which later enables code generation.

To illustrate the use of *match*, let us implement a simple cache mechanism that accepts requests to read and write actions, and responds with the last-written action, beginning with *Left*. Let us represent the request protocol using the data type *Req*.

```
data Req = Read | Write Action
```

Evidently, *Req* is finitely enumerable since it has only three possible values: *Read*, *Write Left*, and *Write Entered*. Hence we may use *match* on a stream *req* :: *Stream Req* as follows.

```
 …
resp ← req `match` λcase
  Read → state
  Write x → val x
state ← Left `fby` resp
 …
```

We shall use ellipses (…) in the code to hide the parts that are not relevant to the point being made. The response stream *resp* is defined by matching against the request stream *req*, where the second argument is a lambda-expression which pattern matches on its argument. We write **λcase** instead of $\lambda x \rightarrow$ **case** x **of**…[2].

The combinator *match* allows us to leverage the benefits of pattern matching in Haskell (such as variable binding, wild cards, guards, etc.) to generate code with simpler branching operators in the target language. For example, the definition of *resp* which pattern matches on *req* in the previous example, generates the following C code.

```
switch (req) {
  case READ      → resp = …
  case WLEFT     → resp = …
  case WENTERED → resp = …
}
```

The cases are representative of the C values generated for the Haskell values of type *Req*.

A pattern match performed using *match* must handle all possible cases, and is enforced by the Haski compiler. If we leave out one of the cases in the above example, the Haski compiler throws an error such as the following—with line-numbers!

```
ghci> compile ...
*** Exception: Cache.hs:(20,18)-(21,22):
Non-exhaustive patterns in case
```

Nodes. The stream *req* in the previous example has not been defined locally, and we wish for it to be a variable which can be substituted for by different contexts. *Nodes* allow us to define subprograms that abstract over stream expressions such as *req*, and thus enable an external caller to supply them. In Haski, nodes are written as Haskell functions, as shown below.

```
cache :: Stream Req → Haski (Stream Action)
cache = node "cache" $ λreq → mdo
  resp ← req `match` λcase
    Read → state
    Write x → val x
  state ← Left `fby` resp
  return resp
```

A node is created using the *node* combinator by providing a name string and a function as arguments.

```
node :: (Arg a, Box b) ⇒ String → (a → b) → (a → b)
```

The name string is used to identify a node uniquely during compilation, and the function defines the body of the node. The type constraints *Arg* and *Box* together ensure that the function $a \rightarrow b$ accepts streams as arguments and produces a stream result in the *Haski* monad, i.e., has a type of the shape *Stream* $a' \rightarrow$ *Stream* $b' \rightarrow \ldots \rightarrow$ *Haski* (*Stream res*).

Notice that the function which defines a node itself need not be inside the *Haski* monad as *Haski* (*Stream* $a' \rightarrow$ *Stream* $b' \rightarrow$

[1] Enabled by the RecursiveDo extension

[2] Enabled by the LambdaCase extension

.... → *Stream res*). This allows for a more natural type to be assigned to a node, and for them to be called and used as regular Haskell functions without any special combinators. For example, we may map over a list of streams as *mapM cache* (*requests*:: [*Stream Req*]) to generate a list of responses, each corresponding to a call of the node *cache*.

Compiling the node *cache* generates code which resembles the following in C.

```
typedef unsigned short Enum;
struct cache_mem {Enum action; };
Enum cache_step (struct cache_mem * self, Enum req) {
  ...
  return resp;
}
```

We shall return to the specifics later, but for now we simply observe that the node *cache* is compiled to a C function with an additional argument *self*. This argument maintains the internal state of the returned stream, which in this case is the last-written action. Also note that both the types (*Req* and *Action*) have been compiled to values of type *Enum*, which represents a small positive integer—a simplifying assumption made for all finitely enumerable types.

Primitive types and operators. The Haski compiler supports standard primitive types of fixed size such as *Bool*, *Int*, etc.

```
instance HT Bool where ...
instance HT Int where ...
  -- similarly for other primitive types
```

The luxury of pattern-matching is limited to finitely enumerable types. Now suppose that we wish to adapt our cache example to a read and write integers instead of actions. Integers are not considered to be finitely enumerable for practical reasons, which means that we cannot use a Haskell data type with an integer in it for pattern matching. Instead, we must separate the request from the integer *payload* into two separate streams as follows.

```
data Req_i = Read | Write
cache_i :: Stream Req_i → Stream Int → Haski (Stream Int)
cache_i = ...
```

To program streams whose types are not finitely enumerable, we resort to the primitive operators supported by the compiler. Haski supports a fixed set of operators that are recognized by the target environment. These operators are overloaded when possible (e.g., +, ∗, etc.) and provided separately otherwise (e.g., *gtE*).

```
(+) :: Stream Int → Stream Int → Stream Int
(*) :: Stream Int → Stream Int → Stream Int
gtE :: Stream Int → Stream Int → Stream Bool
...
```

Sampling operators. In addition to primitive operators, Haski also supports *sampling* operators called *when* and *merge* (from Lustre) for projecting and combining streams.

```
when  :: FinEnum b ⇒ Stream a → (Stream b, b) → Stream a
merge :: FinEnum a ⇒ Stream a → (a → Stream b) → Stream b
```

The operator *when* allows us to project streams to slower ones: the stream s_1 `` `when` `` (s_2, x) produces the value of s_1 only when the value of s_2 is x. Operator *merge*, on the other hand, is a restrictive version of *match* that requires the streams returned by the function argument to be mutually complementary (i.e., at most one stream must produce a value at a time). As we will see in the next section, *merge* is in fact used to implement *match*.

Labeling primitives. Streams which contain sensitive information can be labeled with a sensitivity level. Labeled streams are given the type *LStream a*, and may be understood as streams wrapped in a secure container whose access is controlled using specific primitives. A stream can be labeled and unlabeled using the primitives *label* and *unlabel* respectively, and the label of a stream can be queried using the *labelOf* primitive.

```
label   :: Label → Stream a → Haski (LStream a)
unlabel :: LStream a → Haski (Stream a)
labelOf :: LStream a → Haski Label
```

To understand the use of these primitives, let us implement a new version of the *cache* node where the request and response have been labeled. One reason to do this may be because we wish to keep the actions of a user of our system confidential. To implement the same behavior as before, we must now use the labeling primitives explicitly to label and unlabel the streams.

```
secCache :: LStream Req → Haski (LStream Action)
secCache = node "secCache" $ λreq_l → do
  resp ← unlabel req_l ≫= cache
  ℓ ← labelOf req_l
  resp_l ← label ℓ resp
  return resp_l
```

The code above unlabels the stream req_l as *unlabel* req_l. This raises the senstivity level of the program *secCache* to the label of req_l (also known as *tainting*), which forces all subsequently labeled streams (like $resp_l$) to be at least as sensitive as req_l. The sensitivity level of the program is then used by an administrator to enforce security policies on the program during compilation—as we shall see in Section 5.

3 Overview of Haski compiler

Haski at its core is an embedding of Lustre in Haskell with support for IFC. This means that Haski enables the use of Haskell as a host language to write Lustre programs. A Lustre program, much like Haski, is a system of stream bindings accompanied by a collection of nodes invoked by them. Compiling a Haski

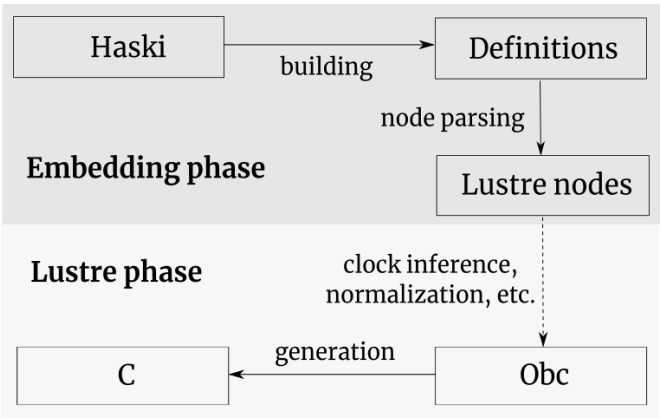

Figure 2. Phases of eDSL compilation. The dashed arrow denotes a sequence of well-known compilation passes used to compile Lustre nodes [Biernacki et al. 2008].

program first builds a Lustre program, and then compiles it to C—thus generating low-level code as in the examples of the previous section.

The compilation function *compile*:: $HT\ a \Rightarrow$ *Haski* (*Stream a*) $\rightarrow$ *IO* () compiles a Haski program and generates C code as a side-effect. Compilation builds a "main" node for the given program, which then acts as the point of invocation for the entire program. Note that the program is restricted to producing an output whose type satisfies the *HT* constraint. This means that, although the program may use any Haskell types, its result must be of a type supported by the target language. This restriction, in combination with similar type constraints on the combinators, ensures that the use of Haskell's features that are not supported by the target environment (such as higher-order functions) are "evaluated away" during compilation time.

The compilation of a Haski program is achieved in two phases (see Figure 2): the Embedding phase constructs a list of Lustre nodes from a Haski program, and the Lustre phase then compiles the nodes to C functions. The first phase is implemented using a combination of deep and shallow embedding techniques, and consists of the compilation passes *building* and *node parsing*. The second phase, on the other hand, transforms Lustre nodes to C functions via an intermediate object-oriented language called Obc. This phase involves a sequence of compilation passes such as clock inference, normalization and scheduling, that are well-known in Lustre compilers [Biernacki et al. 2008].

The Lustre phase is implemented using a modular clock-directed compilation approach that is well-studied and has even been formally verified [Auger et al. 2012; Bourke et al. 2017]. We implement the passes in this phase by repeatedly traversing the abstract syntax tree of Lustre nodes and annotating it with the result of each phase (following Najd and Jones [2017]). Our implementation of this phase is a straightforward adaptation of earlier work, and we do not discuss the details in this paper. Instead, we focus on the implementation details of the first phase, which also forms the basis for the IFC enforcement.

```
data HaskiSt = HaskiSt { defs :: [Def], ... }
type Haski   = State HaskiSt

data Def where
  Let :: HT a ⇒ Var a → Stream a → Def
  Arg :: HT a ⇒ Node → Var a → Stream a → Def
  Res :: HT a ⇒ Node → Var a → Stream a → Def

data Stream a where
  Var    :: HT a ⇒ Var a → Stream a
  Val    :: HT a ⇒ a → Stream a
  Fby    :: HT a ⇒ a → Stream a → Stream a
  When   :: (FinEnum a) ⇒ Stream a
            → (Stream b, b) → Stream b
  Merge  :: (FinEnum a) ⇒ Stream a
            → Vec (Stream b) (Size a) → Stream b
    -- plus primitive operators

type Var a = String
type Node = String
class (Bounded a, Enum a) ⇒ FinEnum a where
  type Size a :: Nat
```

Figure 3. Types used to implement *Haski*

4 Haski as a Lustre Embedding

During the building pass, each line of a Haski program written using one of the combinators builds a corresponding intermediate *definition* under the hood of the *Haski* monad. These definitions are then parsed to construct a complete Lustre program in the node parsing pass. The purpose of this section is to describe the implementation of the building pass, and outline the action performed by the node parsing pass.

4.1 Building Recursive Definitions

The streams defined in the *Haski* monad are collected as a list of definitions. When run with an appropriate initial state, a Haski program produces a list of definitions which correspond to components of Lustre nodes. Definitions are denoted by the *Def* data type, and expressions by *Stream* (see Figure 3). A definition may be a simple binding that binds a variable with a stream expression (*Let*), or an argument (*Arg*) or result (*Res*) of a node call.

The program *alt* from Section 2 builds the following definitions under the hood of the *Haski* monad.

Let "x" ((*Val Left*) `Fby` v_y)
Let "y" ((*Val Entered*) `Fby` v_x)

where v_x = *Var* "x" and v_y = *Var* "y". We use the same variables names as in the original program for readability, but this can also be implemented automatically with some compiler support [Mista and Russo 2020].

Let us now turn to the implementation of combinators in the *Haski* monad. The combinator *fby* is implemented using the *letDef* combinator as follows.

```
fby :: HT a ⇒ a → Stream a → Haski (Stream a)
fby x s = letDef (Fby x s)
```

The combinator *letDef* is in turn implemented by adding a *Let* binding with a fresh variable name to the list of definitions in the *Haski* monad.

```
letDef :: Stream a → Haski (Stream a)
letDef s = do
  x ← freshVar
  addDef (Let x s)   -- updates state ('defs')
  return (Var x)
```

It returns the variable in place of the original stream expression, thus replacing any use of the expression in later definitions with this variable. Returning a variable is the key to enabling recursive definitions without sending the Haski compiler into an infinite loop.

As *fby*, the implementation of *match* also builds definitions containing expressions under the hood, but is slightly more involved since *match* is derived from other expressions. We discuss this next.

4.2 Building Pattern Matching Definitions

The combinator *match* is overloaded in its function argument by the class *Streams* which has the following instances.

```
class Streams b where
  match :: (FinEnum a) ⇒ Stream a → (a → b) → Haski b
instance Streams (Stream b) where ...
instance Streams b ⇒ Streams [b] where ...
instance (Streams b, Streams c) ⇒ Streams (b, c) where ...
  -- similarly for other "containers"
```

The overloading allows the *matching function* $a \to b$ to return multiple streams, such as lists or tuples of streams. In this section, we shall discuss the implementation of the instance *Streams* (*Stream b*). We skip the remaining instances since their implementation is mostly mechanical component-wise applications of *match*.

The combinator *match* provides a convenient interface for defining streams using the more fine-grained sampling operators *When* and *Merge*. For instance, the stream *resp* in the *cache* example from earlier defined using *match* on *req*, builds the following definition.

```
Let "resp" (v_req `When` [
  v_state          `When` (v_req, Read)
  , (Val Left)     `When` (v_req, Write Left)
  , (Val Entered) `When` (v_req, Write Entered)
])
```

When can be understood as a projection of a stream using another stream: the expression v_{state} `When` (v_{req}, *Read*) produces the value of v_{state} when the value of v_{req} is *Read*, and nothing otherwise. In the *Merge* expression above, the vector (written using list notation) contains a stream for each possible value of v_{req}. For every observed value of v_{req}, *Merge* produces the value of the corresponding stream in the vector. The use of *When* ensures that the branches of *Merge* are mutually complementary, which, as mentioned earlier in Section 2, is a restriction that is required of *Merge*.

Now consider implementing the instance *Match* (*Stream b*), where *match* has the type *FinEnum a* ⇒ *Stream a* → (*a* → *Stream b*) → *Haski* (*Stream b*). The matching function $a \to$ *Stream b* is expected to return an expression for every possible value of type *a*. To achieve the semantics of *match* illustrated above, we must implement *match* using *Merge*. But notice that *Merge* requires a *vector* argument of type *Vec* (*Stream b*) (*Size a*) instead of a function, where *Size a* denotes the number of values that inhabit the type *a*. Using a vector forces a *Merge* expression to provide as many stream expressions as the number of values in the type *a* by construction, and thus enables the generated code to also inherit this property. This brings us to the question of implementing *match*: how must we construct a vector of streams from a function which returns them?

The solution to this problem is provided by the *FinEnum* class, which requires all its instances to be both bound and enumerable. Being bound and enumerable means that we could enumerate all the values of an instance type. Additionally, *FinEnum* is also finitely bound by the type family *Size*, which provides a type-level natural number of kind *Nat*. This enables us to enumerate the values as a vector of values, instead of a list of values. Let a function *enumerate* which does this be defined by the following class.

```
class FinEnum a ⇒ Enumerable a (n :: Nat) where
  enumerate :: Vec a n
```

Let us defer its implementation for the time being and simply assume that *enumerate* :: *Vec a* (*Size a*) returns all the values of type *a*.

Since the domain of the matching function is finitely enumerable, we can use *enumerate* to generate all possible arguments to the function. Moreover, we can also apply the function to the enumerated arguments to extract all possible results of the function. Thus we have a way to extract all the stream expressions returned by the function! This behavior is implemented by the following function—named after *"The Trick"* in partial evaluation [Jones et al. 1993].

```
theTrick :: FinEnum a ⇒ (a → b) → Vec b (Size a)
theTrick f = fmap f as
  where as :: Vec a (Size a) = enumerate
```

Equipped with *theTrick*, we implement the desired implementation of *match* as follows.

```
instance Streams (Stream b) where
  match s f = letDef $
  let body  = theTrick f
      whens = theTrick (λx → flip When (s, x))
    in Merge s (zipWith ($) whens body)
```

We first construct the vector which contains the streams on each branch of *Merge* in *body* :: *Vec* (*Stream b*) (*Size a*), and then insert the *When* expressions by zipping it (by application) with *whens* :: *Vec* (*Stream b* → *Stream b*) (*Size a*).

Recollect from earlier that the matching function is enforced to handle all the possible cases of its argument. We do not need any additional checks to enforce this behavior because this is already the case! If the function does not handle all possible cases, the invocation of the function *theTrick* by the compiler crashes with a `Non-exhaustive patterns` error—which, lucky for us, is exactly what we need!

It remains to implement *enumerate*, which is straightforward induction on the *Nat* parameter as follows [3].

```
instance Enumerable a 1 where
  enumerate = [minBound]
instance (Enum a, Enumerable a n, n′∼n + 1)
  ⇒ Enumerable a n′ where
  enumerate = succ (head ts) : ts
    where ts :: Vec a n = enumerate
```

The first value in the vector is constructed using *minBound* and the remaining elements are constructed by applying *succ* on the previous value. These functions are provided by the *Bounded* and *Enum* classes, respectively.

4.3 Building Nodes from Functions

As observed earlier, nodes are Haski subprograms that abstract over streams. Nodes are given a more liberal type which allows them to be regular Haskell functions that need not be defined inside the *Haski* monad. But this creates a challenge: how do we compile a Haskell function which represents a *Haski* node to a data representation of a Lustre node? Moreover, we cannot have a simple *Def* constructor that corresponds to a node call, since *Haski* nodes are not called with a special combinator.

To solve this problem, we first note that result of a node is always in the *Haski* monad. When applied, if we "register" each argument of a node call as a separate definition in the *Haski* monad, then we could recover the complete call in a later pass (node parsing) which acts on the aggregated list of definitions. The idea is to build definitions for a node when it is called, such that the definitions retain sufficient information for the node parsing pass to identify both *the node and its call*. For instance, we wish to build the following definitions for the call *prevAct* ← *cache* (*Val Entered*).

[3] Instance *Enumerable a* 1 requires the {-# OVERLAPPING #-} pragma.

```
class Arg a where
  argDef :: Node → a → Haski a
class Res a where
  resDef :: Node → a → Haski a
instance Arg (Stream a) where ...
instance (Arg a, Arg b) ⇒ Arg (a, b) where ...
instance Res (Stream a) where ...
```

Figure 4. Interface used to register a node call

```
Arg "cache" "arg_1" (Val Entered)
  Let "resp" (v_arg_1 `Merge` [..])
  Let "state" ((Val Left) `Fby` v_resp)
Res "cache" "prevAct" v_resp
```

The body of the *cache* node (containing *Let* definitions) is inlined at the call site by substituting its argument with a fresh variable (v_{arg_1}) instead of the actual argument *Val Entered*. From this invocation, we may recover both the body of the *cache* node and its invocation which defines *prevAct*—which is precisely the job of the node parsing pass. Multiple invocations cause the body to be inlined multiple times, but the parsing pass simply ignores them if a node with a specific name has already been encountered.

Since functions may be partially applied, the arguments must be registered as they are received. Moreover, once all the arguments have been provided the resulting stream must be registered as one resulting from a node call. To achieve this, we shall wrap the function used to create a node inside another function which has the same type, but is also equipped with the ability to register the arguments and the result. This sneaky behavior is implemented by the *node* combinator.

The functions *argDef* and *resDef* (see Figure 4) provide an interface for registering arguments and result of a node. The instances *Arg* (*Stream a*) and *Res* (*Stream a*) allow a stream to be registered as an argument or a result respectively. Their implementation is similar to *letDef*. Additionally, a pair of arguments can also be registered by applying *argDef* on both components of the pair. As we shall see shortly, this instance has to do with registering multiple arguments.

The combinator *node* is implemented by "boxing" the given function using a class *Box* which is overloaded in the return type of the function. It has two instances, *Box* (*Haski b*) for the base case where the function receives a single argument, and *Box* (*b* → *c*) for the inductive case where the function receives more than one argument.

```
class Box b where
  node :: Arg a ⇒ Name → (a → b) → (a → b)
instance (Res b) ⇒ Box (Haski b) where
  node name f = λe → do
    x′ ← argDef name e
```

```
    r ← f x'
    r' ← resDef name r
    return r'
instance (Arg b, Box c) ⇒ Box (b → c) where
  node name f = curry (node name (uncurry f))
```

In the base case instance *Box* (*Haski b*), the function f has the type $a \rightarrow$ *Haski b*. To box this function, we register the argument using *argDef* and call the function with the result of the registration. This substitutes the occurrences of the argument in the body of the function with the stream returned by *argDef*. Finally we register the result of the function using *resDef* and return the corresponding definition.

For the inductive case, observe that we need to box a function $f :: a \rightarrow (b \rightarrow c)$, and the instance declaration provides us instances of *Arg b* and *Box c* as the induction hypotheses. Additionally, we are also given an instance *Arg a* by the declaration of the function *node*. The instances *Arg a* and *Arg b* yield an instance for *Arg* (a, b). Thus, using instances *Arg* (a, b) and *Box c*, we can box the function f by currying it, and then uncurrying back to return the desired result.

5 Information-Flow Control

Haskell is well-known for providing information-flow control (IFC) through security libraries. These libraries ensure that code written using their API does not reveal secrets to unauthorized parties. Many of the existing (monadic) security libraries (e.g., SecLib [Russo et al. 2008], LIO [Stefan et al. 2011b], MAC [Russo 2015], and HLIO [Buiras et al. 2015]) are designed for writing secure code. In this work, however, we consider a different scenario where *we would like to extend an already existing DSL to provide IFC security while minimizing changes to existing code.* Following this goal leads us to the design of an IFC enforcement where security checks are performed at code-generation time rather than at runtime (like in LIO) or type-checking (like in MAC). In this section, we give a brief overview of IFC and explain the design choices of our IFC enforcement for Haski.

5.1 Security lattices

IFC policies enforced by Haski are specified by a security lattice [Denning and Denning 1977], which defines a partial order between security levels (*labels*). These labels represent the sensitivity of program inputs and outputs and the *order* between them dictates which flows of information are allowed in a program. Concretely, we write $\ell_1 \sqsubseteq \ell_2$ if data at security level ℓ_1 can flow to data ℓ_2 according to the security lattice. For example, the classic two-point lattice $\mathscr{L} = (\{L, H\}, \sqsubseteq)$ classifies data as either *public* (L) or *secret* (H) and only prohibits sending secret inputs into public outputs, i.e., $H \not\sqsubseteq L$.

```
  -- Labeled streams
data LStream a
  -- Manipulation of labeled streams
labelOf   :: LStream a → Label
label     :: Label → Stream a → Haski (LStream a)
unlabel   :: LStream a → Haski (Stream a)
  -- Current label
getLabel  :: Haski a → Haski Label
  -- Label creep avoidance
toLabeled :: Haski (Stream a) → Haski (LStream a)
```

Figure 5. IFC interface for Haski

5.2 Enforcement design

We design a coarse-grained IFC enforcement [Vassena et al. 2019], where developers only provide label annotations to security-relevant streams—rather than labeling every stream in a program. A labeled stream of type *LStream* is implemented by associating a stream expression with its label as follows.

```
data LStream a = LStream {getLabel :: Label, getStr :: Stream a}
```

The type *LStream* acts as an opaque container since its implementation is not exposed to the programmer. For instance, the labeled stream *LStream Halexa* (*val* 42) is a constant stream that is confidential to the smart house controller *Halexa* .

Figure 5 shows Haski's IFC interface, which provides primitives to manipulate labeled streams while avoiding information leakage. Function *labelOf* obtains the label associated with a labeled stream. To understand the rest of the primitives, we need to introduce the concept of a *floating label*.

Every line in the *Haski* monad is associated with a special label known as the floating label (denoted by *fl*), which "floats above" the label of any observed stream during program execution and thus represents an upper-bound on the sensitivity of all the streams in scope. The floating label is tracked in the state of the *Haski* monad:

```
data HaskiSt = HaskiSt {defs :: [Def], fl :: Label, ...}
```

In order to enforce IFC policies, Haski regulates the interaction between *Haski* computations and labeled streams. *Haski* computations cannot write and read labeled streams directly, but must use the primitives in Figure 5. Let us discuss the implementation of these primitives next.

5.3 Implementing the labeling primitives

The labeling primitives create and read labeled streams in compliance with specific security rules to avoid information leakage [Bell and La Padula 1976].

The primitive *label* labels a stream with the given label and does not affect the floating label of the program. Its implementation ensures that the desired label ℓ is at least the floating label of the program, i.e., $fl \sqsubseteq \ell$, thus enforcing a *no write-down*

$$<C_1, I_1> \sqsubseteq <C_2, I_2> \iff (C_2 \Rightarrow C_1) \wedge (I_1 \Rightarrow I_2)$$

$$<C_1, I_1> \sqcup <C_2, I_2> \iff <C_1 \wedge C_2, I_1 \vee I_2>$$

$$<C_1, I_1> \sqcap <C_2, I_2> \iff <C_1 \vee C_2, I_1 \wedge I_2>$$

$$\bot \equiv <True, False> \qquad \top \equiv <False, True>$$

Figure 6. DC-labels semantics

policy. Intuitively, *label* creates a labeled stream as long as the decision to do so depends on less sensitive data. For example, given $fl = L$, the invocation *label* H s (for some $s :: Stream\ Int$) is legal since $fl \sqsubseteq H$. This means that a program which has read sensitive data cannot write public information in an attempt to leak it. If this criteria is not met, *label* inserts an error using *fail* in the *Haski* monad, thus crashing compilation.

The primitive *unlabel* acts as the dual of label and extracts the stream underlying a labeled stream. Unlike *label*, however, *unlabel* never crashes compilation and always succeeds. Instead, the invocation *unlabel* s_l raises the floating label of the program to $fl \sqcup \ell$.

Haski, as any other floating-label based IFC systems, suffers from the label creep problem. Unlabeling sensitive streams raises the floating label of the program, and hence a program which reads many sensitive streams risks raising its level to a point where it may not be able to produce any observable result. This problem is remedied using the *toLabeled* primitive, which addresses it by (i) creating a separate context where some sensitive computation can take place and (ii) restoring the original floating label afterwards.

The argument of *toLabeled* is a sensitive computation of type *Haski* (*Stream a*), that cannot return its result to the outer context—since that would be a leak. Instead, *toLabeled* wraps the result in a labeled stream using the floating label resulting from the execution of the sensitive computation. Unlike *unlabel*, *toLabeled* produces a labeled stream of type *LStream a* and its invocation does not affect the floating label. An invocation of *toLabeled* never fails.

5.4 Running programs securely

DC-labels. Haski uses DC-labels [Stefan et al. 2011a], which is an expressive label format that can capture the security concerns of principals. DC-labels are pairs of confidentiality and integrity policies, noted $< C, I >$ where C is the confidentiality policy and I is the integrity one. Both policies are positive propositional formulas in conjunctive normal form (CNF), where propositional constants represent *principals*. We assume that operations on formulas always reduce their results to CNF. For simplicity, we focus on confidentiality since the integrity part comes as a dual of it. Given two confidentiality policies C_1 and C_2, we interpret $<C_1, I> \sqsubseteq <C_2, I>$ as: C_2 is at least as confidential as C_1. For instance, $<Halexa \vee$ Octavius, $I> \sqsubseteq <$Octavius, $I>$, which means that data readable by either *Halexa* or the Octavius is less confidential than data readable only by the Octavius. In contrast, given two integrity policies I_1 and I_2, we interpret $< C, I_1 > \sqsubseteq < C, I_2 >$ as: I_1 is more trustworthy than I_2, i.e., there are more principals taking responsibility for the data labeled with I_1 than in I_2. For instance, $<C$, Octavius $\wedge$ *Halexa*$> \sqsubseteq <C$, *Halexa*$>$, which means that *Halexa* and the Octavius are jointly responsible for the data, which is more trustworthy than data only vouched by Octavius. Figure 6 presents the formalization of operations we will use in the rest of this section together with the definition of $\sqcup$ and $\sqcap$ in the security lattice. With DC-labels in place, we can associate the different components of our system to different principals, thus enabling them to impose different restrictions on the confidentiality and integrity of data.

Configuring security policies. A Haski program that returns a stream (labeled or not) can be run using the *runAs* function on behalf of a principal. This function is intended to be used by an administrator who compiles a Haski program and assigns the right privilege to it—we assume that the administrator is part of the trusted computing base. Function *runAs* is defined as follows:

```
class IsStream f where
  runAs :: Haski (f a) → Principal → Haski (Label, Stream a)
```

The result of the *Haski* $(f\ a)$ argument is overloaded in f to allow for both labeled and unlabeled streams to be returned. The *Principal* argument is used to set the initial floating label of the *Haski* computation and denotes the source of authority, i.e., the entity, that this program represents. For example, *runAs prog* `"Halexa"` runs a computation on behalf of Halexa with the DC-label $<Halexa, Halexa>$. As a result, any stream that is labeled by *prog* will contain *Halexa* in both the confidentiality and integrity components of its label—which means that the stream is confidential to *Halexa*, and also that *Halexa* has contributed to its content.

The *runAs* function returns a label that corresponds to the final floating label of the computation joined with the label of its result, along with the result that it returns. The label is intended to be used by the administrator to enforce application-specific security policies. Observe that the result is an unlabeled stream. This is due to the fact that *runAs* is run by the administrator, i.e., a person that we trust, so there is no need to protect the resulting stream by labeling it.

We implement the *runAs* function using the *toLabeled* primitive. This is because *toLabeled* allows us to create a separate context for the program to be run in, and, as observed earlier, restores the floating label of the administrator prior to execution. Restoring the floating label of the administrator allows the administrator to run programs on behalf of various principals without getting tainted by them. Here is the

```
type Status = Maybe Action
data WindowOp = Skip | Open | Close
halexa :: Stream Int → LStream Status
          → Haski (LStream WindowOp)
halexa = node "halexa" $ λtemp stat_l → do
  isHot ← letDef $ temp `gtE` 30
  toLabeled $ do
    stat ← unlabel stat_l
    pastAct ← (stat `match` mkReq) ≫= cache
    recentAct ← stat `match` (maybe pastAct val)
    dec ← recentAct `match` λcase
      Left → val Close
      Entered → ifte isHot (val Open) (val Skip)
    return dec
  where
    mkReq :: Status → Stream Req
    mkReq Nothing = val Read
    mkReq (Just x) = val (Write x)
```

Figure 7. Implementation of *Halexa*

Stream instance which implements *runAs* for computations that return expressions.

```
instance IsStream Stream where
  runAs prog princ = do
    (LStream ℓ res) ← toLabeled $ do
      setLabel (newDCLabel princ princ)
      prog
    return (ℓ, res)
```

Function *setLabel* can only be used by the administrator and it is part of the trusted computed base, i.e., it is present in the IFC interface exposed to developers. The function *newDCLabel* creates a label from the given principal by using it for both the confidentiality and integrity components.

The instance for the case of labeled expressions is implemented in turn using the above instance by simply unlabeling the result.

```
instance IsStream LStream where
  runAs prog princ = runAs (prog ≫= unlabel) princ
```

The intended effect of this implementation is for the resulting label to be $\ell \sqcup fl$, where *fl* is the floating label of *prog* at the end of its execution, and ℓ is the label of its result.

6 A Sample Application

In this section we illustrate the structure of the *Halexa* application and its security policy in Haski. The purpose of our application is to make a decision on opening a window, based on the current temperature in the house and the status of the user Octavius. *Halexa* is expected to open the window when the temperature in the room is over 30°C provided Octavius is at home. If Octavius is not home, however, *Halexa* must close the window regardless of the temperature. We consider the status of Octavius sensitive information and thus we require *Halexa* to confine the status and any information derived from it. That is, the status cannot be used to build streams less sensitive than the DC-label <Octavius, Octavius>.

We model *Halexa* as a node which accepts two streams as arguments (see Figure 7): one of type *Stream Int* for the temperature reading, and another of type *LStream Status* for a labeled stream of notifications which notify *Halexa* about the actions of Octavius. The notifications specify whether Octavius has left (*Just Left*), entered (*Just Entered*), or that there is no change in status (*Nothing*). In response, the node returns a stream of instructions denoted by *Stream WindowOp* which instructs whether the window should be opened (*Open*), closed (*Close*), or whether nothing should be done (*Skip*). In essence, we implement *Halexa* using the *toLabeled* primitive to unlabel the labeled stream $stat_l$, thus ensuring that *Halexa* does not read its contents.

To understand the logic of the implementation, notice that a status stream *stat* need not contain any update in Octavius's action since it may be *Nothing*. Hence it is up to us to compute the whereabouts of Octavius from the *most recently observed action*. We compute this in the stream *recentAct* as follows: if the current value of *stat* is *Nothing* then use the last available action of the user (given by *pastAct*), else simply use the action given by *stat*. The stream *pastAct* retains the last action of the user using the *cache* node from earlier. Finally, we define a decision stream by matching on the *recentAct* stream, which produces the desired result. The combinator *ifte* is simply a shortened version of a *match* expression which pattern matches on *True* and *False*.

An administrator who wishes to run *Halexa* must provide the appropriate input streams to the node and assign the right policies using the function *runAs*. One such implementation is the following.

```
admin :: Haski (Stream WindowOp)
admin = do
  temp ← ...
  status ← ...
  status_l ← label ℓ_o status
  (res, ℓ) ← runAs (halexa temp status_l) (principal "Halexa")
  unless (ℓ ⊑ (ℓ_o ⊔ ℓ_h)) (fail "Bad Halexa")
  return res
  where
    ℓ_o = newDCLabel "Oct" "Oct"
    ℓ_h = newDCLabel "Halexa" "Halexa"
```

The security policy *unless*... in *admin* asserts that the resulting label must at most be a combination (⊔) of the labels of *Octavius* and *Halexa*. A simple case of obtaining the inputs would be to simply use fresh variables to define streams *temp*

and *status*, which are then later initiated by the runtime. For a more realistic system, however, we require a way to obtain streams from entities outside of a Haski program. We discuss one possibility to address this requirement via bluetooth in the next section.

7 Reacting to Streams Outside of Haski

A typical IoT application communicates with several other applications and reacts to triggers which may originate from remote devices. To use Haski to build more realistic applications, it is important to enable streams to be provided by external sources. In this section, we consider the case of obtaining streams from remote devices via Bluetooth, which is a common means of communication in low power IoT devices. We manage to run *Halexa* by creating a small C runtime around the code generated by Haski. In essence, the runtime obtains the *temp* and *status* streams from earlier via the Bluetooth Low-Energy (BLE) API of Zephyr OS on the nrf52840DK board using the techniques discussed here with some manual intervention.

7.1 Briefly about Bluetooth Low Energy

The Bluetooth component we target uses the BLE stack on Zephyr OS[4], where the most common way that data flows through a BLE application is through a *Generic Attribute Profile* (GATT) server. Specifically, a device that has some data it wishes to make available to other devices will take the role of a GATT server. It will organise the data it has as *characteristics* that belong to *services*. As an example, a device might expose a biometrics service which in turn exposes the heart rate characteristic and the temperature characteristic.

A remote device that wishes to access or modify these values will take the role of a GATT client. A GATT client will initiate a connection to a GATT server, after which it scans for services and characteristics. Depending on the server configuration the client can update a remote characteristic, read a characteristic or subscribe to be notified about changes to a characteristic.

7.2 Preparing Halexa for foreign streams

A Haski program works on streams, yet the APIs we want to use in Zephyr OS use commands and callback functions. These need to be connected somehow.

For example, the Bluetooth API contains a function called *bt_gatt_subscribe* that is used to register a callback function whenever a message is received from a specified device. In Haski, when we subscribe to a device, we do not provide a callback function, but we receive a Haski stream instead:

$$btGattSubscribe :: DeviceID \to Haski\ (Stream\ a)$$

So, for example, in order to connect the *Halexa* example from the previous section to the devices *tempSensor* and *motionSensor*, we can write the following code:

$$temp \leftarrow btGattSubscribe\ tempSensor$$
$$status \leftarrow btGattSubscribe\ motionSensor$$
$$\ldots$$

The compilation process will then generate an invocation of the C function *bt_gatt_subscribe* in the generated code and registers a callback to the *step* function—which is generated for every node—of *Halexa*. This means that the *step* function is called every time the devices *tempSensor* and *motionSensor* provide an update. Since the *step* function receives two arguments and the devices only produce one of them at a time, the *step* function is called with a default argument for the other. For example, the value of the *status* stream is *Nothing* when *tempSensor* provides an update.

7.3 The Halexa GATT Client

The BLE code that ties together the *Halexa* example with the remote temperature and the motion sensor assumes the role of a GATT client. The GATT client will scan for remote devices by calling the *bt_le_scan_start* BLE API function. The following function signatures have been simplified and rewritten in Haskell notation, and many less interesting functions have been omitted. The actual C versions of the API functions can be found in Appendix A.1.

$$bt\_le\_scan\_start :: ScanParams \to (RemoteDeviceInfo \to Int) \to Int$$

The second argument is a function that will be invoked when a device has been found. Once a remote device is found, a connection will be initiated with *bt_conn_le_create*.

$$bt\_conn\_le\_create :: RemoteAddress \to CreateParams \to ConnectionParams \to Connection \to Int$$

When the connection has been established, we will scan it for the services it exposes. We expect to discover, e.g., the temperature service. To do this, we need to create some discovery parameters and then invoke *bt_gatt_discover*.

$$bt\_gatt\_discover :: Connection \to DiscoverParams \to Int$$

A subexpression of *DiscoverParams* is a function that will be called when a service have been discovered. This function will subscribe to a found service by invoking *bt_gatt_subscribe*. This will make sure that *Halexa* is notified about any changes to the remote temperature value.

$$bt\_gatt\_subscribe :: Connection \to SubscribeParams \to Int$$

The *SubscribeParams* contain a function that will be called every time a notification is received. The function will be invoked with values describing the connection that issued the notification as well as the actual payload.

[4] https://www.zephyrproject.org/

Recollect from earlier that a node in Haski is compiled to *step* function in C which is invoked in response to the availability of its arguments. Compiling *Halexa* from the previous section generates a corresponding step function *halexa_step*. This function has the following signature.

```
Enum halexa_step (struct halexa_mem * self,
                  int temp, Enum motion)
```

In addition to this function, compiling *Halexa* also generates a struct *halexa_mem*, an instance of which is provided as the argument *self* to function *halexa_step*. This argument maintains the internal state of the stream returned by *Halexa*.

```
struct halexa_mem {...};
```

For every call of a node in a Haski program, an instance of such a struct is initialized globally before the first invocation, and passed as an argument to every subsequent invocation of the correponding *step* function. For *Halexa*, initialization is done as follows.

```
/* Global definition */
struct halexa_mem * mem;
...
/* Evaluated by main */
mem = k_malloc (sizeof (struct halexa_mem));
```

Using these definitions we build a function that is registered as a callback to be invoked whenever the BLE application receives, for example, a new temperature reading (as shown below).

```
static u8_t notify_temperature (..., const void * data ) {
  ...
  int * temperature = (int*) data ;
  ...
  halexa_step (mem, *temperature, NOTHING);
  ...
}
```

We invoke the function *halexa_step* with its internal memory *mem*, which stores the internal state of the node. Notice that we pass *NOTHING*, a representation of the corresponding Haskell value, for the status stream here. This is because the function *notify_temperature* is invoked in response to the temperature sensor, which does not provide a status update. A similar callback function must be registered for the *status* stream by invoking *halexa_step* with a default temperature reading.

We emphasize that the small C runtime we implemented here is tailored to BLE and it requires some manual intervention to make the coupling between the generated code by Haski and Zephyr OS's API—we leave as future work to devise an automatic mechanism to do that.

7.4 Going Forward

The attentive reader might have paused to think while reading the previous section. The previous section describes how we compile a synchronous programming language to a target which uses callbacks and events instead of streams. It is not immediately obvious how to do this. This discrepancy leads to the need for manual intervention when connecting the generated code to the outside world via BLE.

There are a few questions that need to be addressed in future work. How is a continuous stream created from the sporadic events given to a callback function by the outside world? How do you compile a Haski node and dynamically register and unregister it as a callback?

We believe nicely generalising this is possible, and leave this and more questions as future work.

8 Related Work

Synchronous languages. The seminal work of Lustre [Caspi et al. 1987] (sometimes called "classical Lustre") shows how a declarative synchronous programming style can benefit from memory and computational time bounds. Lustre's ideas have been applied in a wide-range of scenarios ranging from hardware design (e.g., [Bjesse et al. 1998]) to real-time reactive systems (e.g., [Qian et al. 2015]).

Haski is based on a variation of classical Lustre from Biernacki et al. [2008], the semantics of which has been formalized and verified by Auger et al. [2012] and Bourke et al. [2017]. The main difference between classical Lustre and the variant used by Haski is the absence of the *current* operator and the addition of the *merge* and *reset* operators. For a more detailed discussion on the differences, see Bourke et al. [2017]. Haski does not (yet) implement the *reset* operator.

A notable implementation of Lustre that is closely related to ours is Lucid Synchrone [Caspi et al. 2008]. Lucid Synchrone uses OCaml as the host language and allows a rich programming interface with many higher-order features of OCaml. Unlike Haski, it allows pattern matching on complex data types (e.g., streams of functions) that are not limited to finitely enumerable types. Naturally, the richer features offered by Lucid Synchrone also place higher demands from the runtime system, such as the need for a garbage collector. Haski, on the other hand, targets memory constrained IoT devices and thus strives to keep the runtime system minimal. The code generated by compiling a Haski program can be executed with a fixed amount of memory and does not require garbage collection.

Functional Reactive Programming. Functional Reactive Programming (FRP) [Elliott and Hudak 1997] is a programming style for programming asynchronous reactive systems. Unlike Lustre, it has the convenience of incorporating higher-order functions at the price of possibly introducing memory leaks—as noticed and addressed in subsequent work (e.g., [Bahr et al. 2019; Courtney et al. 2003; van der Ploeg and Claessen 2015]).

Haski does not support higher-order functions as first class values, but enables developers to utilize them to build first-order Lustre programs. The staged programming approach ensures that all higher-order functions are eliminated at compile time, thus removing the need to address space leaks which may be caused by them.

Code generation for C. We are not the first ones to propose an eDSL in Haskell for generating memory safe C code. Closest to our work is Copilot [Pike et al. 2013], an eDSL for stream-based programming for avionics. While Copilot provides similar guarantees on the generated code w.r.t. constant space and execution time, Haski presents a different programming experience (e.g., a monadic interface) as well as IFC security features. Haskino [Grebe and Gill 2016] is an eDSL to write programs to be run in an Arduino board while supporting a light-weight notion of threads. Like Haski, Haskino deploys the generated-C code into a custom made runtime. Feldspar [Axelsson et al. 2010] is a DSL for describing digital signal processing algorithms in Haskell and generate C code. Ivory [Elliott et al. 2015] is an advanced DSL for writing memory-safe C code within Haskell. It uses a simple notion of memory regions and also provides access control security checks to restrict side-effects in the generated C-code.

Language-based security for IoT. Pyronia [Melara et al. 2019] provides access control and IFC for embedded devices written in Python. Pyronia runs under a custom-made runtime responsible to perform system call interposition, call stack inspection, and memory protection. Such modification are required to ensure that Python, where by design data is public, can safely execute and interact with C programs. In contrast, Haskell provides good abstractions to deliver a pure language-based IFC solution [Russo 2015; Russo et al. 2008; Stefan et al. 2011b], which enables Haski to not require special runtimes and run on commodity IoT OSes. SainT [Celik et al. 2018] delivers an static IFC analysis for commodity SmartThings apps. SainT builds an intermediate representation for Groovy (object-oriented) SmartThings programs, where IFC checks are carried out. SainT targets legacy code while Haski provides security by construction using a coarse-grained IFC approach. Hence, SainT needs to extend the semantics of Groovy commands to reason about IFC. Instead, Haski provides modular security types (*LStream*) and primitives (e.g., *label* and *unlabel*) atop of our synchronous language. Velox VM [Tsiftes and Voigt 2018] provides a Scheme virtual machine for constrained devices. Every app run by the VM has an associated *access control* policy file, which is used to restrict apps from accessing sensitive data and resource usage. As future work, Haski could integrate resource usage control as done by Velox VM.

Haskell security libraries. The closest Haskell IFC libraries to our approach are LIO [Russo 2015], HLIO [Buiras et al. 2015], and MAC [Stefan et al. 2011b]. Our approach to enforce IFC at compile-time leads us to a new design space, where our API is a simplified version of the LIO's one due to executing the analysis at compile-time. More specifically, LIO takes an extra parameter in *toLabeled* to avoid leakage via labels [Buiras et al. 2014], which Haski does not suffer from by taking an static (compile-time) approach. Compared with HLIO and MAC, Haski is static but does not rely on Haskell's type-system for security checks but rather on the Haski compiler. Generally speaking, Haski's IFC API is a static, simplified, version of LIO's API while not going all the way to HLIO or MAC—it is something in between.

9 Final Remarks

We have presented Haski, a Haskell eDSL for writing software in embedded devices. Haski generates C code with memory consumption guarantees as well as information-flow security thanks to many program analyses realized by the compiler. We showcase that Haski programs can be easily integrated with a realistic runtime like the BLE in Zephyr OS. We expect this work to be a foundation to build IoT applications that leverage, not only BLE, but most of the underlying embedded OS functionality while providing security properties. Furthermore, we leave as future work to adapt our eDSL to allow users to be "in the loop" when relaxing IFC restrictions, e.g., to enable opening windows when the user is not home or to allow sending occupancy information to a security monitor firm. The Haski core development[5] (excluding the BLE runtime) currently consists of 2621 lines of Haskell code.

Acknowledgments

This work was funded by the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023) and by the Swedish research agency Vetenskapsrådet under the project SyTeC (Ref. 2016-06204). We would like to thank Nordic Semiconductor for providing us with the nRF52840DK boards, and the anonymous reviewers at Haskell '20 for their feedback on this paper.

A Appendix

A.1 BLE API function prototypes in C

For brevity, the following API signatures were rendered as Haskell's type signatures in the paper. Below, we show the complete Zephyr OS' API signatures for the methods described in the paper.

```
int bt_le_scan_start (const struct bt_le_scan_param * param,
  bt_le_scan_cb_t device_found)
int bt_conn_le_create (const bt_addr_le_t * peer,
  const struct bt_conn_le_create_param * create_param,
  const struct bt_le_conn_para * conn_param,
  struct bt_conn * conn)
int bt_gatt_discover (struct bt_conn * conn,
```

[5] https://github.com/OctopiChalmers/haski

*struct bt_gatt_discover_params * params*)

int bt_gatt_subscribe (*struct bt_conn * conn*,
*struct bt_gatt_subscribe_params * params*)

References

Cédric Auger, J. L. Colaco, Grégoire Hamon, and Marc Pouzet. 2012. A Formalization and Proof of a Modular Lustre Compiler.

Emil Axelsson, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Bo Lyckegård, Anders Persson, Mary Sheeran, Josef Svenningsson, and András Vajda. 2010. Feldspar: A domain specific language for digital signal processing algorithms. In *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*. 169–178.

Patrick Bahr, Christian Graulund, and Rasmus Ejlers Møgelberg. 2019. Simply RaTT: a fitch-style modal calculus for reactive programming without space leaks. *Proc. ACM Program. Lang.* 3, ICFP (2019).

David E. Bell and L. La Padula. 1976. *Secure Computer System: Unified Exposition and Multics Interpretation.* Technical Report MTR-2997, Rev. 1. MITRE Corporation, Bedford, MA.

Elisa Bertino and Nayeem Islam. 2017. Botnets and Internet of Things Security. *IEEE Computer* 50, 2 (2017), 76–79.

Dariusz Biernacki, Jean-Louis Colaço, Gregoire Hamon, and Marc Pouzet. 2008. Clock-directed modular code generation for synchronous data-flow languages. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*. 121–130.

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware Design in Haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*.

Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. 2017. A formally verified compiler for Lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 586–601.

Pablo Buiras, Deian Stefan, and Alejandro Russo. 2014. On Dynamic Flow-Sensitive Floating-Label Systems. In *IEEE Computer Security Foundations Symposium, CSF*. 65–79.

P. Buiras, D. Vytiniotis, and A. Russo. 2015. HLIO: Mixing Static and Dynamic Typing for Information-Flow Control in Haskell. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP '15)*. ACM.

Paul Caspi, Grégoire Hamon, and Marc Pouzet. 2008. Synchronous functional programming: The lucid synchrone experiment. *Real-Time Systems: Description and Verification Techniques: Theory and Tools. Hermes* (2008), 28–41.

Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. 1987. Lustre: A Declarative Language for Programming Synchronous Systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*.

Z. Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick D. McDaniel, and A. Selcuk Uluagac. 2018. Sensitive Information Tracking in Commodity IoT. In *USENIX Security*.

Antony Courtney, Henrik Nilsson, and John Peterson. 2003. The Yampa arcade. In *Proc. of the ACM SIGPLAN Workshop on Haskell*. 7–18.

D. E. Denning and P. J. Denning. 1977. Certification of Programs for Secure Information Flow. *Commun. ACM* 20, 7 (July 1977), 504–513.

Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming*. 263–273.

Trevor Elliott, Lee Pike, Simon Winwood, Patrick C. Hickey, James Bielman, Jamey Sharp, Eric L. Seidel, and John Launchbury. 2015. Guilt free ivory. In *Proc. of the ACM SIGPLAN Symposium on Haskell*. 189–200.

Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. 2016. Security Analysis of Emerging Smart Home Applications. In *IEEE Symposium on Security and Privacy, SP*. 636–654.

Mark Grebe and Andy Gill. 2016. Threading the Arduino with Haskell. In *Trends in Functional Programming - 17th International Conference, TFP*. 135–154.

Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320.

P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. 2011. Data Representation Synthesis. In *Proc. ACM Conference on Programming Language Design and Implementation*.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Inc., USA.

Marcela S. Melara, David H. Liu, and Michael J. Freedman. 2019. Pyronia: Redesigning Least Privilege and Isolation for the Age of IoT. *CoRR* abs/1903.01950 (2019). arXiv:1903.01950 http://arxiv.org/abs/1903.01950

Agustin Mista and Alejandro Russo. 2020. BinderAnn: Automated Reification of Source Annotations for Monadic EDSLs. In *21st International Symposium on Trends in Functional Programming, TFP*.

Shayan Najd and Simon Peyton Jones. 2017. Trees that Grow. *Journal of Universal Computer Science* 23, 1 (2017), 42–62.

Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming, ICFP*.

Matthew Pickering, Gergo Érdi, Simon Peyton Jones, and Richard A. Eisenberg. 2016. Pattern synonyms. In *Proc. of the 9th International Symposium on Haskell, Haskell 2016*.

Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. 2013. Copilot: monitoring embedded systems. *ISSE* 9, 4 (2013), 235–255.

Jie Qian, Jing Liu, Xiang Chen, and Junfeng Sun. 2015. Modeling and Verification of Zone Controller: The SCADE Experience in China's Railway Systems. In *1st IEEE/ACM International Workshop on Complex Faults and Failures in Large Software Systems, COUFLESS 2015*. 48–54.

A. Russo. 2015. Functional Pearl: Two Can Keep a Secret, if One of Them Uses Haskell. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM.

A. Russo, K. Claessen, and J. Hughes. 2008. A library for light-weight information-flow security in Haskell. In *Proc. ACM SIGPLAN symposium on Haskell (HASKELL '08)*. ACM.

A. Sabelfeld and A. C. Myers. 2003. Language-Based Information-Flow Security. *IEEE J. Selected Areas in Communications* 21, 1 (Jan. 2003), 5–19.

Roei Schuster, Vitaly Shmatikov, and Eran Tromer. 2018. Situational Access Control in the Internet of Things. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS*.

D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. 2011a. Disjunction Category Labels. In *Proc. of the Nordic Conference on Information Security Technology for Applications (NORDSEC '11)*. Springer-Verlag.

D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. 2011b. Flexible Dynamic Information Flow Control in Haskell. In *Proc. of the ACM SIGPLAN Haskell symposium (HASKELL '11)*.

Nicolas Tsiftes and Thiemo Voigt. 2018. Velox VM: A safe execution environment for resource-constrained IoT applications. *J. Netw. Comput. Appl.* 118 (2018).

Atze van der Ploeg and Koen Claessen. 2015. Practical principled FRP: forget the past, change the future, FRPNow!. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming, ICFP*. 302–314.

Marco Vassena, Alejandro Russo, Deepak Garg, Vineet Rajani, and Deian Stefan. 2019. From fine- to coarse-grained dynamic information flow control and back. *Proc. ACM Program. Lang.* 3, POPL (2019), 76:1–76:31.

Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl A. Gunter. 2018. Fear and Logging in the Internet of Things. In *25th Annual Network and Distributed System Security Symposium, NDSS*.

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a Promotion. In *Proc. of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '12)*. ACM.