Flask: Staged Functional Programming for Sensor Networks

Geoffrey Mainland Greg Morrisett Matt Welsh

Harvard School of Engineering and Applied Sciences

{mainland,greg,mdw}@eecs.harvard.edu

Abstract

Severely resource-constrained devices present a confounding challenge to the functional programmer: we are used to having powerful abstraction facilities at our fingertips, but how can we make use of these tools on a device with an 8- or 16-bit CPU and at most tens of kilobytes of RAM? Motivated by this challenge, we have developed Flask, a domain specific language embedded in Haskell that brings the power of functional programming to sensor networks, collections of highly resource-constrained devices. Flask consists of a staging mechanism that cleanly separates node-level code from the meta-language used to generate node-level code fragments; syntactic support for embedding standard sensor network code; a restricted subset of Haskell that runs on sensor networks and constrains program space and time consumption; a higher-level "data stream" combinator library for quickly constructing sensor network programs; and an extensible runtime that provides commonly-used services.

We demonstrate Flask through several small code examples as well as a compiler that generates node-level code to execute a network-wide query specified in a SQL-like language. We show how using Flask ensures constraints on space and time behavior. Through microbenchmarks and measurements on physical hardware, we demonstrate that Flask produces programs that are efficient in terms of CPU and memory usage and that can run effectively on existing sensor network hardware.

Categories and Subject Descriptors D.3.3 [*Software*]: Programming Languages

General Terms Languages, Design

Keywords Meta programming

1. Introduction

Sensor networks consist of ensembles of small, cheap devices whose power lies in numbers. The TelosB mote, a typical member of this class of devices, has a 16-bit TI MSP430 CPU, 10K of RAM, 48K of program flash memory and a 250 kbps 802.15.4 radio. The vision for sensor networks is that these devices can be deployed cheaply and in large numbers, facilitating the collection of data on a scale that was simply not possible before. The challenge is to develop the algorithms and software necessary to realize this goal. Flask is a domain specific language embedded in Haskell that seeks to address this challenge by making the type of high-level, reusable abstractions commonly developed in the functional programming community available in the sensor network domain.

Flask's programming model inspired by the large body of work on Functional Reactive Programming (FRP) (Elliott and Hudak 1997; Nilsson et al. 2002; Pembeci et al. 2002; Hudak et al. 2003) Unfortunately, this work does not translate directly to the sensor network domain because sensor nodes simply do not have enough RAM or ROM to run a full Haskell environment—GHC compiles "Hello, world!" to a binary weighing in at more than 350K, and the Hugs runtime is over 500K. Even the experimental jhc compiler produces an 8K binary for this same program, never mind the heap space required for even a moderately complex application. To target a TelosB-class device, it is necessary to forgo conveniences such as garbage collection and closure allocation, but *only for code running on the nodes themselves*. Flask provides a staging mechanism that maintains a clean separation between node-level code and the meta-language used to generate node-level code fragments, so programmers can still use the full power of Haskell as a meta-language when programming sensor networks. Node-level code can be written in a language called Red, which is syntactically equivalent to Haskell but disallows closures and recursive data types, thereby eliminating arbitrary allocation. The combination of staging plus a restricted object language form the basis for our approach to adapting the ideas from FRP to work in the sensor network domain.

In addition to a staging mechanism and Red, Flask consists of general syntactic support for embedding standard sensor network code; a higher-level "data stream" combinator library for quickly constructing sensor network programs; and an extensible runtime that provides commonly-used services. Although Flask is geared towards streaming data applications, it also provides facilities for building higher-level abstractions, which we use in Section 6 to write a concise, network-wide fold operator that then forms the basis for a compiler from a SQL-like query language to native sensor network binaries. The version of Flask detailed in this paper is a complete rewrite of an earlier system, written in OCaml, that was the basis for previous unpublished work.

Flask's contributions are as follows:

- **A design for embedding object languages in Haskell:** Flask builds on our previous work on quasiquotation (Mainland 2007) to provide quasiquoting for both nesC (Gay et al. 2003), a C-based language commonly used for sensor network programming, and Red. This minimizes the syntactic burden of embedding object languages in Haskell. We also show how to leverage the Haskell type system to type object language terms.
- **Seamless integration of existing sensor network code into a functional programming environment:** Flask places nesC on equal footing with Red in that signal combinators such as `map` can apply a nesC function to signal as easily as applying a Red

function. Anywhere a programmer can write Red, he can also write nesC.

- **Red, a restricted subset of Haskell that has a well-defined semantic relationship to its parent language and compiles to efficient node-level code:** Red is syntactically identical to Haskell 98, although it lacks support for a few Haskell 98 features such as type classes. Additionally, the Red type-checker constrains Red code so that both data types and functions are non-recursive and so that closures cannot be allocated. Therefore, all Red functions are total and allocation is bounded. In this setting, call-by-value evaluation is equivalent to call-by-need evaluation, so we choose the former for efficiency reasons although Haskell programmers can still informally reason about Red code as if it obeyed the same evaluations semantics as Haskell. The constraints imposed by the Red type checker also ensure that Red can be compiled to efficient sensor network code, as we show in Section 7.

The rest of this paper is organized as follows. We begin in Section 2 by providing background and motivating our solution. Section 3 gives an overview of Flask and its design. In Section 4 we discuss the implementation of Flask's combinators, the Red compilation strategy, and Flask's interface to nesC. We next evaluate Flask in three key areas; we evaluate Flask's usefulness as a sensor network programming language via a case study of a geophysical monitoring system implemented in Flask in Section 5; in Section 6 we show that Flask is suitable for building high-level abstractions by developing a compiler that translates statements in a SQL-like language to executable code; and in Section 7 we present microbenchmarks comparing Flask programs to hand-written nesC variants and describe a Flask program running on a sensor network testbed consisting of approximately 160 TelosB motes. Section 8 describes related work, and Section 9 presents future work and concludes.

2. Background and Motivation

Sensor networks present the tantalizing possibility of building better, cheaper scientific instruments that can collect data that was previously impractical to acquire. One of the authors has worked closely with vulcanologists to deploy sensor networks that collected real-world seismic data at Reventador (Werner-Allen et al. 2006) and Volcán Tungurahua (Werner-Allen et al. 2005a), both active volcanoes in Ecuador. While the results from these trials were promising, the associated software required several man-years of effort and resulted in a system that was tied to a particular deployment. Part of the problem is the tools available to would-be sensor network programmers; most applications are written in nesC, a component-based dialect of C, and involve a lot of low-level event-driven code. Even more challenging, programmers must grapple not only with the constraints imposed on individual devices, but they must somehow get these devices to work together using a low bit-rate, unreliable radio connection.

When forced to spend most of one's time staring at leaves, it is particularly challenging to see the forest, and a natural consequence is that many sensor network programs are tailored to specific deployments and contain an abundance of application-specific code rather than reusable abstractions. The sensor network community recognizes this challenge and has produced a number of systems that seek to relieve the pain of programming sensor networks (Madden et al. 2005; Bakshi et al. 2005b; Cheong et al. 2003; Whitehouse et al. 2005; Kothari et al. 2007; Newton et al. 2007; Levis et al. 2005; Sorber et al. 2007; Gnawali et al. 2006). Many of these tools focus on particular problem domains, limiting their generality, or only run in simulation or on PDA-class devices, limiting their applicability.

Flask is targeted at an emerging class of sensor network applications that deal primarily with (possibly) high-rate streaming data, of which the Reventador deployment is a prime example. Other examples include structural monitoring (Xu et al. 2004) and acoustic signal processing (Ali et al. 2007). As such, Flask is designed around a stream-based data model in which data originates from sensors or incoming radio packets and flows through a series of operators that can transform, filter, store or transmit data. FRP's dataflow model and the high-level, re-usable abstractions it provides are a natural fit for this domain. We show the power of these types of abstractions in Section 6. Using a functional language also tends to reduce the amount code required to express program logic, a claim that is supported by our experience using Flask, and which we describe in Section 5. The FP approach of defining small, reusable and composable combinators, which we follow in Section 3, is one of driving forces behind this reduction in code size, and we used it to our advantage throughout the development of both Flask and the applications we describe in the rest of the paper. Purity also allows Flask to potentially take advantage of algebraic optimizations that are not valid for impure code. In light of these advantages, it was natural for us to ask how one can augment an existing language, in particular Haskell, with enough mechanism so that we can bring the power of functional programming to sensor networks.

The development of Flask was driven by three observations. First, there will always be devices with severely limited capabilities similar to those of the aforementioned TelosB mote; they will simply become smaller and cheaper. If we can program these devices effectively, then as they become smaller and cheaper they will also become more practical to deploy in large numbers. Thus the problem of programming ensembles of TelosB-like devices will not disappear or become obsolete due to advances in hardware. Second, for one to have any hope of running functional programs on such resource constrained devices, it is clear that the code that runs on the sensor network hardware and the code that expresses the reactive part of programs must be separately staged (Taha 1999; Taha and Sheard 2000). This separation between the meta-language, which in our case is Haskell, and the object language, which can be either nesC or a restricted subset of Haskell called Red, enables Flask to tightly constrain the run-time behavior of sensor network code while allowing the programmer to stitch together functionality using the full power of Haskell. Third, the code that makes use of the features functional programmers take for granted—higher-order functions, closures, polymorphism, etc.—is exactly the code that does the stitching. By simultaneously paring down the object language and offering powerful "stitching" combinators, Flask enables functional programming on sensor networks.

3. Overview

We begin by motivating and justifying the basic design of Flask. Our discussion is oriented around a simplified version of the earthquake detection algorithm used by Werner-Allen et al. (2006). Although simplified, this application demonstrates the main features of Flask and also represents the core logic of the more complex volcano monitoring application described in Section 5. We begin by writing the detector using Arrowized FRP (AFRP), as embodied in the Yampa Haskell library (Nilsson et al. 2002). Although device constraints prevent us from running the program we develop here on a TelosB-class sensor network, this exercise shows how one might write a sensor network program in a functional style. We subsequently use Flask to re-express this program in a manner that allows it to be efficiently executed on a real sensor network.

3.1 Sample Application: Earthquake Detection in FRP

The main logic of the earthquake detection application is shown in Figure 1. The detection algorithm works by first separately

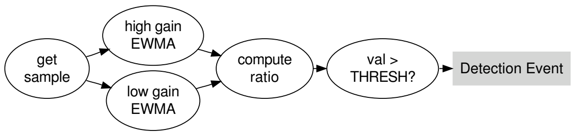

Figure 1: **Conceptual depiction of earthquake detection.**

applying high-pass and low-pass filters to the input signal. If the ratio between the values output by the high-pass and low-pass filters exceed a threshold, then a detection event has occurred.

It is straightforward to write this detector using Arrowized FRP (AFRP). AFRP uses the SF type to represent Signal Functions, functions from one continuous stream of time-varying values to another continuous stream of time-varying values. Also provided is the Event type, which models a discrete-time signal. The detect function takes parameters for the high- and low-pass filters and a detection threshold and returns a stream function from seismic readings to earthquake detection events. It has the following type:

```
detect :: Double → Double → Double
          → SF Double (Event ())
```

Implementing the high- and low-pass filters requires remembering some state representing the past history of the seismic signal. These filters can be implemented using an exponentially weighted moving average (EWMA) which computes the moving average of a value from past observations by weighting the observation at time $t - n$ by the factor α^n for $0 < \alpha < 1$. An EWMA estimate is updated with a new value by multiplying the old moving average by $1 - \alpha$ and adding the new value weighted by the factor α. To write an ewma signal function, we need to lift a pure function that computes a new moving average when given both the previous output and an input; this function should have type $\beta \to \alpha \to \beta$. Yampa provides such a combinator, which also takes an initial "previous" value:

```
sscan :: (β → α → β) → β → SF α β
```

With sscan we can write our ewma signal function as follows:

```
ewma :: Double → SF Double Double
ewma α =
  sscan (λx_old x → α ∗ x + (1 − α) ∗ x_old)
        x0
```

Here x_0 is a value appropriate for the initial state of the ewma filter, e.g., the value output by a quiescent seismometer.

Before we can finish writing detect, we need some additional basic arrow combinators to lift a pure function to a signal function; compose signal functions; and pair the results of applying two signal functions.

```
pure    :: (a → b) → SF a b
(>>>)   :: SF a b → SF b c → SF a c
(&&&)   :: SF a b → SF a c → SF a (b, c)
```

We also need one more Yampa-specific signal function that performs edge detection on a signal of Boolean values:

```
edge :: SF Bool (Event ())
```

With these functions in hand, the translation from Figure 1 to Haskell is quite direct:

```
detect :: Double → Double → Double
          → SF Double (Event ())
detect low high thresh =
  ewma high &&& ewma low
  >>>
  pure (λ(hi, lo) → hi / lo > thresh)
  >>>
  edge
```

In fact, most sensor network applications follow a common pattern: collect data, perhaps perform some simple computation or filtering, and then forward the collected data over a low-power radio back to a base station for further analysis. The forwarding step often involves a routing protocol which forms a spanning tree connecting the sensor network nodes to the base station via a multi-hop path. Sometimes locally-connected values are not forwarded immediately, but held for a period of time so they can be aggregated with data from other nodes before being sent upstream, thus saving bandwidth and energy. While not universal, this class of application is widespread enough in the sensor network domain that it makes sense to focus on providing powerful tools for programming in this style.

Using FRP's built-in combinators, we have implemented the core logic of an earthquake detector in a very small amount of code. Furthermore, the program logic is evident from the source text and closely follows our conceptual depiction of the detection algorithm. Despite these advantages, there is one glaring problem with using FRP directly to program sensor networks: nodes just don't have the necessary horsepower or memory. However, there is hope.

3.2 Overview of Flask

Notice that in our detect function the flow of data through the program that we arranged by applying the &&&, >>>, and pure combinators *does not change* during program execution. It is the point-wise manipulation of values that must be executed dynamically; the data flow configuration is static. Flask exploits this observation by explicitly staging the static and dynamic behavior of dataflow programs (Taha 1999). Haskell serves as Flask's meta language, and its full power is available for expressing how data flow graphs can be constructed using FRP-style combinators. Flask also requires that point-wise manipulation of signal values is always done in a restricted object-language. Because the dynamic behavior of the program is expressed only through a restricted object language, Flask programs can run on even highly resource-constrained devices like the TelosB mote.

The arrow abstraction on which AFRP is based (Hughes 2000, 2004) does not make a distinction between meta and object level programs, which prevents us from using it directly in Flask; the Arrow class method pure, of which we have seen a special case, has type Arrow a ⇒ (b → c) → a b c, which allows one to lift any pure Haskell function to operate point-wise over signals. In Flask, only object level functions can be lifted to operate on signals, so the Arrow class is not what we want. However, we do reuse some of the good ideas from arrows which we note as they are developed.

An obvious choice for an object language is nesC (Gay et al. 2003). Not only is it restricted enough to run on sensor nodes, but it allows us to interface with a large body of existing nesC and TinyOS (Hill et al. 2000) code into which the sensor network community has invested significant effort. For example, the detect function needs a data source. In our case this data source is a custom built analog-to-digital converter with a driver written in nesC. It would be both impractical and unnecessary to rewrite this body of code. Thus, providing nesC as an object language lets us easily integrate the existing driver into Flask.

We address the syntactic challenge of mixing nesC and Haskell code using GHC's quasiquoting facility (Mainland 2007). Quasiquoting allows nesC to be written directly in a Haskell source file using standard nesC syntax instead of forcing the programmer to construct a Haskell representation of the corresponding abstract syntax. We provide a quasiquoter for nesC functions, cfun, that automatically performs the translation from nesC concrete syntax to abstract syntax when a Haskell file is compiled. Values that exist at the meta level can also be spliced into object level code using antiquoting. Returning to our detect example, we use our quasiquoter cfun to write a staged version of the pure function that tests the ratio between the high- and low-pass filters as follows:

```
thresh_test thresh =
  [$cfun | bool test(double hi, double lo) {
             return hi/lo > $flo:thresh;
         }
   |]
```

Not only do we make a clear distinction between the meta level function thresh_test and the object level function `test`, but also between the dynamic values `hi` and `lo` and the static value thresh. Because thresh is static—it does not change as the detection program runs—it exists at the meta level as a parameter to the Haskell function f. The syntax `$flo:thresh` tells the quasiquoter to splice in the meta level floating point variable thresh as a constant in the quoted nesC function. We elaborate on quasiquoting in Section 4.

3.3 Red: A Restricted Object Language

While it is necessary to directly use nesC as an object language to provide access to the large body of existing sensor network code, it is still inconvenient in many respects. For example, nesC's lack of support for tuples and algebraic data types is particularly painful in the dataflow setting, e.g., the &&& combinator naturally wants to use a tuple. In general, nesC is not a good fit as an object language for FRP-style programming. To help bridge the semantic gap between functional programming and low-level sensor network code, we provide a second object language, called Red, that is syntactically equivalent to Haskell 98.

By default, the Red type checker enforces three additional constraints over standard Haskell 98: recursive data types are disallowed, closures may not be allocated, and recursive functions are disallowed. This ensures that Red code is terminating and can only perform bounded allocation. Because functions written in Red are total, they can be implemented using a call-by-value evaluation strategy, but the programmer can reason about them as if they were evaluated using the same call-by-need strategy used for the meta language, Haskell. By using Red, the programmer gets even stronger guarantees than if he were to use nesC, which provides an unrestricted "escape hatch" back into an effectful, unbounded world. Given terminating object code that performs only bounded allocation, Flask combinators will only produce object terms that perform bounded allocation and terminate. Another advantage of using Red is that it opens up a number of opportunities for optimization; nesC code can have side effects, making algebraic optimizations and equational reasoning difficult.

Using Red, we can rewrite the staged version of our threshold test as follows:

```
thresh_test :: Double → Exp
thresh_test thresh_val =
  [$exp | λ(hi, lo) → hi / lo > $flo:thresh |]
```

In addition to staging all computations by explicitly separating the object and meta levels, Flask's design differs from FRP in its signal abstraction. Like earlier incarnations of FRP, Flask provides signals rather than AFRP's signal functions. Unlike FRP, Flask's signals are always discrete. Discrete signals better reflect the asynchronous event-based computation model in TinyOS, the sensor network platform on which Flask is built. A Flask signal carrying values of type α has type S α.

```
constant :: N α → S β → S α
  map    :: N (α → β) → S α → S β
 filter  :: N (α → Bool) → S α → S α
  &&&    :: S α → S β → S (α, β)
  >>>    :: (S α → S β) → (S β → S γ)
            → (S α → S γ)
  >>>=   :: S α → (S α → S β) → S β
 merge   :: S α → S α → S α
fromJust :: S (Maybe α) → S α
  edge   :: S Bool → S ()
```

Figure 2: **Basic Flask signal combinators.**

Figure 2 shows some of the basic Flask signal combinators. We can now write a version of detect that will run on real sensor network hardware:

```
detect :: Double → Double → Double
      → S Double → S ()
detect low high thresh =
  (λsig → ewma high sig &&& ewma low sig)
   >>>
  map [$exp | λ(hi, lo) → hi / lo > $flo:thresh |]
   >>>
  edge
```

The result of running a combinator like detect is a residual object level program, which is then efficiently compiled to nesC and run on sensor network hardware. In the following section we discuss the quasiquoting facility and how we use it to embed object languages in Haskell. We then discuss Flask's signal model and address practical issues like concurrency and persistent state. In Section 5 we develop a significant application, and in Section 6 we show how to build higher level abstractions with Flask.

4. Implementation

The primary challenge in implementing Flask is providing support for object languages distinct from Haskell. Most existing metaprogramming environments make the assumption that the meta and object level languages are identical (Taha and Sheard 2000; Sheard and Peyton Jones 2002), which is clearly not appropriate for our domain. Unlike these systems, Flask must produce a residual program that is not a Haskell term requiring a Haskell runtime for execution, but that can be efficiently implemented on severely resource constrained hardware. Designing mechanisms to support distinct object languages in Haskell requires addressing two key issues: syntax and typing. We next describe how to add this support to Haskell.

4.1 Supporting object languages in Haskell

The key to adding syntactic support for object languages has already been described: quasiquoting. In previous work, we added quasiquoting support to the GHC compiler (Mainland 2007), which will appear in the next release (6.10) of GHC. This extension was motivated by the current application, although it has many other uses as well. For completeness we review how quasiquoting works, and then we describe how it is used in Flask.

Quasiquoting allows programmers to use domain specific concrete syntax to construct Haskell terms. The implementation in GHC makes use of existing Template Haskell (Sheard and Peyton Jones 2002) machinery to convert domain specific syntax to Haskell terms by calling a programmer-specified *quasiquoter*. The quasiquoter is called at compile time, before type checking is performed, so there is no possibility of compiling ill-typed Haskell.

The syntax for quasiquotation is similar to the syntax used by Template Haskell for staged computations (Sheard and Peyton Jones 2002). Whereas Template Haskell quotes a Haskell expression using bracket-bar pairs, e.g., `[|1 + 2|]`, a quasiquotation uses an additional dollar sign and identifier following the initial open bracket. Consider again the final staged version of the detect function previously defined:

```
detect :: Double → Double → Double
        → S Double → S ()
detect low high thresh =
   (λsig → ewma high sig &&& ewma low sig)
   >>>
   map [$exp | λ(hi, lo) → hi / lo > $flo:thresh |]
   >>>
   edge
```

GHC transforms the quasiquoted term passed as an argument to map into a Haskell term by calling the quasiquoter bound to the Haskell variable exp and passing it the string contained between the brackets, in this case `"\(hi, lo) -> hi/lo > $flo:thresh"`. The quasiquoter returns Haskell abstract syntax for the term that is denoted by this concrete syntax. Flask provides a number of quasiquoters for both NesC and Red, allowing expressions, declarations, types, etc. from both languages to be quasiquoted.

For quasiquotation to be truly useful, quasiquoters must support splicing Haskell terms into quoted terms. This is accomplished via antiquotation, seen in the detect example where it causes, at the time the quoted expression is evaluated, the value bound to the Haskell variable thresh to be inserted as the second argument to the comparison in the abstract syntax tree for the quasiquoted Red term. As a second example, the following function pair takes two arguments, each representing abstract syntax for a Red type, and returns abstract syntax for the Red tuple type formed from the two arguments:

```
pair :: Type → Type → Type
pair ty1 ty2 = [$ty | ($ty:ty1, $ty:ty2) |]
```

The syntax $ty : and $exp : is specific to the Red quasiquoter, and tells it to splice existing Red abstract syntax trees into the parse tree, in this case abstract syntax nodes representing a Red type and a Red expression, respectively.

4.2 Typing object language terms

Although quasiquoting solves the syntactic problem of how to embed an object language in Haskell, it does not help with the problem of assigning accurate types to object-level terms. For example, quasiquoted Red expressions all have the Haskell type Exp, which does not constrain the *object level* type of the expression. There are several options when dealing with this issue. First, we could choose to ignore the problem and delay type-checking until the residual node-level program is compiled. But then we couldn't even give useful types to Flask's stream combinators; the best we could do for Flask's map signal combinator would be to give it the type Exp → S → S, where the constructor S no longer carries a type parameter. Another option is to fully integrate Haskell's type system with the object language's type system, a heavyweight alternative that also requires substantial modifications to a Haskell compiler. A middle-ground solution would be to reflect the object language's types as Haskell types and then encode the object language's abstract syntax using GADTs (Xi et al. 2003; Jones et al. 2006). While this encoding suffices for simple languages, for even a moderately complex language there is a subtle problem: the GADT carries only the type of the term it represents, and does *not* carry a typing environment. One might be able to work around this limitation by either asserting that all terms are typed in a single, fixed typing environment, or by requiring that terms only be produced in a monad that carries the typing environment; even then, there are subtle issues involving scope, variable ordering, and alpha-conversion that are difficult or impossible to encode without resorting to relatively heavyweight tricks. We explored both of these possibilities, but found that in practice they do not work well.

The solution we chose is to wrap untyped object level terms behind a type constructor N carrying a single type parameter that reflects the object level term's type at the Haskell type level. This representation serves two main purposes. First, it allows us to type object level terms and assign more appealing types to combinators as shown Figure 2. Second, it allows us to write pure code without requiring that object level terms be typed in a fixed type environment by wrapping object level type checking in a monadic action that is only run when the Flask program is residualized. This means that the type N α serves only as a contract that the object level term it represents should resolve to object-level code with type α. A Flask program containing ill-type object level terms can still run, but it will fail to produce a residualized object level program if any such ill-typed object term is used.

To type check object terms of type N α we need a way of *reifying* the type α so it can be represented as an object level type. This type is then used to type check the object term when the program is residualized. Type reification is accomplished with the type class Reify:

```
class Reify α where
  reify :: α → Type
```

The member function reify takes a parameter of type α, which serves only to fix the type being reified, and returns the object level type corresponding to α. To illustrate further, we give the instance declaration for the types Int and $\forall\alpha\ \beta.(\alpha, \beta)$:

```
instance Reify Int where
  reify _ = [$ty | Int |]
instance ∀α β.(Reify α, Reify β) ⇒ Reify (α, β) where
  reify _ = [$ty | ($ty:ty_α, $ty:ty_β) |]
    where
      ty_α = reify (⊥ :: α)
      ty_β = reify (⊥ :: β)
```

As mentioned, the N type constructor encapsulates a monadic action. When this action is run, it checks that the object term has the proper type. Additionally, object terms are *hash consed* (Goto 1974) to preserve sharing. Preserving sharing is important: without doing so, it is all too easy to write code that ends up requiring exponential space. Monadic actions are run in the Flask monad, which tracks the state necessary to support hash consing, contains the top-level type environment, and records the information necessary to produce a residual nesC program. We elide the details of hash consing in the remainder of the section to simplify the discussion.

In general, we want to support a number of object languages and allow terms from any such language to be represented as a value of type N α. Therefore, we define a second type class, LiftN, that allows object terms to be *lifted* to values of type N α.

```
class LiftN η α where
  liftN :: η → N α
```

We can lift Red expression as follows:

$$\begin{array}{l} \textbf{instance}\ \forall\alpha.(\mathsf{Reify}\ \alpha) \Rightarrow \mathsf{LiftN}\ \mathsf{Exp}\ \alpha\ \textbf{where} \\ \quad \mathsf{liftN\ e} = \mathsf{N\ (checkExp\ e\ ty)} \\ \qquad \textbf{where} \\ \qquad\quad \mathsf{ty} :: \mathsf{Type} \\ \qquad\quad \mathsf{ty} = \mathsf{reify}\ (\bot :: \alpha) \end{array}$$

The LiftN type class provides an added bonus: we can more directly parametrize Flask combinators by a variety of different object languages. Given LiftN, we can assign a more general type to Flask's map combinator:

$$\mathsf{map} :: \mathsf{LiftN}\ \eta\ (\alpha \rightarrow \beta) \Rightarrow \eta \rightarrow \mathsf{S}\ \alpha \rightarrow \mathsf{S}\ \beta$$

This more flexible type for map lets the programmer write:

$$\begin{array}{l} \mathsf{inc} :: \mathsf{S\ Int} \rightarrow \mathsf{S\ Int} \\ \mathsf{inc} = \mathsf{map}\ [\$\mathsf{exp} \mid (+1) \mid] \end{array}$$

without having to explicitly insert calls to liftN.

4.3 Signals in Flask

Manipulating a Flask signal can require the implementation to perform an effectful computation. For example, nesC has a standard interface, `ADC`, for components that produce sensor data. Creating a Flask signal from such a source requires generating the proper nesC code to instantiate the sampling component and hook it to downstream operators. However, these operations do not need to be performed until the program is residualized, so we reuse the technique we applied to our representation of object terms and wrap the monadic action that encapsulates these effects in a data constructor S.

Preserving sharing is even more vital in the signal setting. Consider the clock and adc combinators:

$$\begin{array}{l} \mathsf{clock} :: \mathsf{Int} \rightarrow \mathsf{S}\ () \\ \mathsf{adc}\ \ :: \mathsf{String} \rightarrow \mathsf{S}\ () \rightarrow \mathsf{S\ Double} \end{array}$$

The clock combinator takes a value in milliseconds and produces a signal of unit impulses at the specified rate. The adc combinator takes the name of a nesC component implementing the ADC interface, a stream of impulses to drive sampling, and returns a stream of values sampled at the given rate. Both combinators return a monadic action wrapped in the S type constructor. When this monadic action is run at program residualization time, it generates nesC code to instantiate the proper component and call downstream operators when an event, such as the arrival of a sample, occurs. Because Flask exposes signals directly, signals can be named—that is, they can be bound to a Haskell variable. Returning to the detect example, if sharing were not preserved, the &&& combinator would duplicate the signal sig twice. This being the case, applying detect to a signal created with the adc combinator would cause the resulting residual program to attempt to instantiate the nesC component twice because the monadic action created by calling adc would be executed twice, once for each branch of the computation.

As with the N data type, we address this problem by hash consing signals. When the clock combinator is invoked, it returns a wrapped monadic action that first checks to see if a clock signal running at the given rate has already been created by a previous monadic action; only if no such action has yet been performed will it instantiate the signal itself. Similarly, let-binding the Haskell expression `adc "Seismometer"` generates only a single instance of the seismometer component in the residualized program no matter how many times the binding is used at the meta level.

4.4 Stateful computations and concurrency

In addition to the sharing issue, Flask must deal with practical issues related not to its embedding in Haskell, but to nesC and TinyOS. TinyOS is a sensor network programming framework written in nesC that requires programs to be written using an asynchronous event model. For example, the `ADC` interface exposes two functions, a *command* `getData` that tells the component implementing the interface to acquire a single data sample, and an *event* `dataReady` that the consumer of the `ADC` interface must provide and that acts as a callback for an acquired sample. This pattern pervades the standard interfaces in TinyOS: operations are not performed by calling a blocking function that returns a result. Instead, one calls a command that enqueues a request and returns immediately, also providing a callback that is invoked when the result of the operation is available. If a signal function invokes such an asynchronous operation, it is possible that a subsequent invocation of the signal function will occur before the first invocation finishes executing. If the signal function needs to maintain some sort of state, this kind of asynchronous operation can easily lead to a race condition.

Avoiding state where possible is clearly preferable, but there are times when state is necessary. Instead of forbidding state, we provide combinators that allows the programmer to limit the scope of stateful computations, thereby also limiting possible bugs like race conditions. Inspired by AFRP, Flask provides an integrate combinator, which is a "map with state":

$$\begin{array}{l} \mathsf{integrate}\ ::\ \mathsf{LiftN}\ \eta\ ((\alpha,\sigma) \rightarrow (\beta,\sigma)) \\ \qquad\qquad \Rightarrow \mathsf{N}\ \sigma \rightarrow \eta \rightarrow \mathsf{S}\ \alpha \rightarrow \mathsf{S}\ \beta \end{array}$$

The second parameter to this combinator is an object level function which takes an input value paired with the current state and returns an output value paired with the updated state. With integrate, we can now complete the Flask implementation of detect by writing the ewma function:

$$\begin{array}{l} \mathsf{ewma} :: \mathsf{Double} \rightarrow \mathsf{S\ Double} \rightarrow \mathsf{S\ Double} \\ \mathsf{ewma}\ \alpha = \mathsf{integrate\ zero} \\ \qquad\qquad\quad [\$\mathsf{exp} \mid \lambda(\mathsf{x}, \mathsf{x}_{\mathsf{old}}) \rightarrow \\ \qquad\qquad\qquad \textbf{let}\ \mathsf{x}' = \$\mathsf{flo}{:}\alpha * \mathsf{x} + \\ \qquad\qquad\qquad\qquad\quad (1.0 - \$\mathsf{flo}{:}\alpha) * \mathsf{x}_{\mathsf{old}} \\ \qquad\qquad\qquad \textbf{in}\ (\mathsf{x}', \mathsf{x}') \mid] \\ \quad \textbf{where} \\ \qquad \mathsf{zero} :: \mathsf{N\ Double} \\ \qquad \mathsf{zero} = \mathsf{liftN}\ [\$\mathsf{exp} \mid 0.0 \mid] \end{array}$$

Note that the initial state to be used, zero, must be given an explicit type signature. This is due to the fact that the Haskell type checker cannot infer the phantom type variable σ, and is a shortcoming of the object language embedding approach we chose.

When a signal function is not synchronous, we can still allow it to use state if we ensure that multiple executions of the signal function are not interleaved. Consider a signal function applied to a signal of sampled data that during its execution calls out to a NesC component to perform an asynchronous operation, e.g., logging a value to flash memory. We would like a mechanism that allows any signal function to use state, but in this particular example we must contend with the possibility that a new sample will arrive before the previous sample is logged. If samples can be processed as fast as they arrive in the steady state, then we could solve the state problem by queueing new input samples until the previous sample has been completely processed. Flask supplies a combinator, loop, that provides this functionality. Inspired by the the ArrowLoop class's loop method, its type is:

$$\begin{array}{l} \mathsf{loop}\ ::\ \mathsf{Int} \rightarrow \mathsf{N}\ \sigma \rightarrow (\mathsf{S}\ (\alpha,\sigma) \rightarrow \mathsf{S}\ (\beta,\sigma)) \\ \qquad \rightarrow \mathsf{S}\ \alpha \rightarrow \mathsf{S}\ \beta \end{array}$$

The loop combinator threads state through an entire signal function, ensuring race-free access even if intermediate stages of the signal function involve asynchronous processing. If a new value ar-

	Flask	Original (NesC)
Main program logic	27	n/a
Command processing	21	368
Fetch protocol	18	190
Eruption detection	26	143
Sample to datastore	n/a	163
NesC components		
Sampling	624	624
Flash storage	750	750
Routing protocol	2217*	1072
Flask wrappers		
Sampling	101	n/a
Flash storage	214	n/a
Sample to datastore	179	n/a

Figure 3: **Breakdown of the Flask and NesC versions of the volcano application in terms of lines of code.** **Represents the Flows protocol in the Flask version and MultiHopLQI plus Drip in the NesC version.*

rives from the input signal of type S α before the previous value has been completely processed by the signal function, it is enqueued in a fixed-sized queue whose depth is given by the first argument to loop.

4.5 Generating Residual Programs

To convert a Flask signal to a residual nesC program, the monadic action representing the signal is run. This action generates two bodies of residual code: a nesC component *configuration*, specifying which nesC components are instantiated and how they are connected, and an intermediate Red program that corresponds to the implementations of the instantiated components' interfaces. This intermediate Red program is composed of type-checked object level terms; Red object terms are incorporated directly, and nesC object terms are imported as foreign function calls. These terms represent the point-wise signal operations that were composed to create the signal being reified.

The intermediate Red program is then type checked again as a whole and elaborated to a typed intermediate language that is a variant of System F_C (Sulzmann et al. 2007). Because closures are disallowed, lambda lifting is trivial to perform. Polymorphic functions are specialized for every type at which they are used, or *statically monomorphized*; this feature is supported by other compilers such as MLton (Weeks et al.). The typed intermediate language is then compiled to nesC code, which is combined with all nesC object terms and output to a nesC *module*. This module implements the component interfaces used in the previously generated configuration. The generated nesC module and configuration are compiled to produce a node-level binary. This binary is the residual program.

5. Application Case Study: Geophysical Monitoring

In this section we take as a case study a Flask application for real-time geophysical monitoring of volcanic activity. This application is based on an original system (Werner-Allen et al. 2006) implemented directly in NesC whose source is publicly available, and doing the port gives us the opportunity to directly compare the complexity and memory footprint of the Flask and NesC implementations. Several extraneous features were left out of the Flask port which are excluded from our comparison here. We chose this system because it represents a fairly complex, real-world sensor network application involving high-data-rate sampling, flash storage, signal processing, and reliable communication. Complete details on the original system can be found in Werner-Allen et al. (2006). The program is fairly sophisticated, involving 32 custom NesC components, in addition to numerous standard TinyOS libraries.

Figure 4 shows the dataflow graph for the complete application. Shown this way, the structure of the code is self-evident, whereas the original NesC implementation is a complex set of interrelated components. It is also clear that the structure of this application cannot be readily described as a simple linear chain of operations; there are multiple branches for control and data flow.

Figure 3 gives a breakdown of the lines of code for the major components of each implementation. The main body of the Flask wiring program is just 27 lines of code. It is difficult to directly compare this to the corresponding NesC application, since the "main structure" is scattered across numerous modules and component wirings. It is clear that using Flask reduces implementation complexity for most components. For example, the Fetch protocol is 190 lines in NesC, but just 18 lines in Flask, a reduction of 90%. The Flows routing code is longer than *MultihopLQI* and *Drip*, the protocols used in the original implementation, but this is not surprising given its increased generality—MultihopLQI and Drip only form spanning trees.

6. Building higher-level abstractions

Part of the appeal of functional programming is the power of the abstraction facilities it provides, such as polymorphism, higher order functions, closures and strong typing. It is this appeal that led us to develop Flask. To demonstrate how Flask leverages these abstraction facilities in the sensor network domain, we show how Flask can express high-level abstractions that describe not just the behavior of a single sensor network node, but the behavior of an entire ensemble of devices.

Before we can proceed, we need combinators that allow Flask signals to be carried between nodes. The Flask runtime provides a flexible routing protocol, called Flows, that allows streams of values to be sent and received over the radio. This runtime functionality is exposed to the programmer by the send and recv combinators, which associate signals with abstract radio *channels*.

```
send :: FlowChannel → S α → S ()
recv :: FlowChannel → S α
```

Although the Flows protocol can be used to implement several kind of routing topologies, the examples in this paper only use it to form spanning trees that connect all participating sensor nodes back to a single root node. In this configuration, each channel corresponds to a separate spanning tree; nodes receive data from their children on the channel, and data sent on the channel is delivered to a node's parent in the spanning tree.

Given send and recv, we continue with two examples of higher level abstractions written using Flask: a network-wide fold combinator, and FlaskDB, a compiler for a simple SQL-like query language.

6.1 A macroprogramming combinator: nfold

Much research in the sensor network community has been geared toward so-called *macroprogramming*, where a single program specifies the operation of an entire ensemble of sensor network nodes (Madden et al. 2005; Bakshi et al. 2005b; Whitehouse et al. 2005; Kothari et al. 2007; Newton et al. 2007; Sorber et al. 2007). By choosing the appropriate primitive operations, Flask allows first-class macroprogramming combinators to be written by the programmer rather than requiring that they be built-in. Our goal is to write a combinator, nfold, that acts much like integrate, but operates on signals received from other nodes as well as node-local signals. The idea is that both a local signal and a signal containing data from other nodes are combined, fed to a stateful signal function, the output of which is then forwarded to the appropriate next

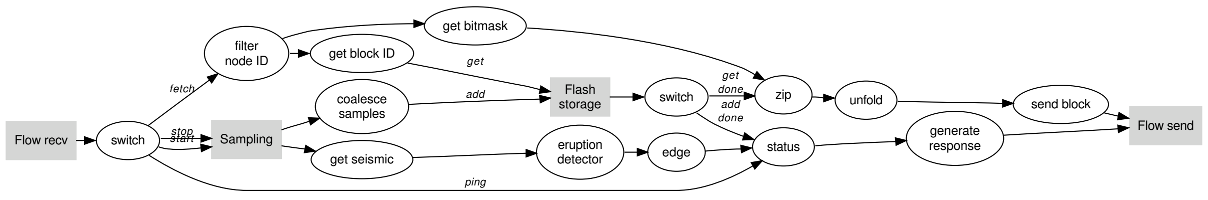

Figure 4: **Dataflow representation of the volcano monitoring application.** *Gray boxes represent components implemented as NesC wrappers; the rest are implemented directly in Flask.*

hop. When this operation is run on the network of sensor nodes, the ensemble will form a spanning tree, aggregate values received from other nodes with locally-generated values, and forward the aggregates up the spanning tree. This combinator can be defined as follows:

$$\begin{aligned}&\mathsf{nfold} :: \mathsf{FlowChannel}\\&\quad\to \mathsf{N}\ \sigma \to (\mathsf{S}\ (\alpha,\sigma) \to \mathsf{S}\ (\alpha,\sigma))\\&\quad\to \mathsf{S}\ \alpha \to \mathsf{S}\ ()\\&\mathsf{nfold\ chan\ zero\ f\ sin} =\\&\qquad\mathsf{merge\ (recv\ chan)\ sin}\\&\quad\ggg\!= \mathsf{loop\ zero\ f}\\&\quad\ggg\!= \mathsf{send\ chan}\end{aligned}$$

That we can so easily write a combinator that operates over an entire sensor network is a testament to the power of the abstractions provided by Flask. The strategy of choosing a small number of appropriate primitive operations and providing powerful facilities for composing operations to produce new first class operations is not unique to Flask; it is a mainstay of the functional programming literature and has resulted in many powerful systems, from parsing combinators (Hutton 1992) to evaluation of financial contracts (Jones et al. 2000) to program testing (Claessen and Hughes 2000). Flask brings the power of this paradigm to bear on sensor network programming.

6.2 Running queries with FlaskDB

Flask is a useful toolkit for building up higher-level sensor network abstractions while providing the efficiency of compiled NesC. In this section, we describe *FlaskDB*, an implementation of the TinyDB (Madden et al. 2005) query system in Flask. Unlike TinyDB, which uses a runtime query processing engine, FlaskDB compiles a TinyDB query into a static, fully-optimized sensor node binary. We demonstrate the operation of the FlaskDB compiler by describing how it compiles the following query:

```
SELECT COUNT(ID), AVG(TEMP) PERIOD 10s
```

This FlaskDB query requests that the attributes `ID` and `TEMP` be collected from every sensor node, and the *aggregate functions* `COUNT` and `AVG` be applied to the attributes `ID` and `TEMP`, respectively, to produce *aggregates*. Instead of blindly sending all attributes up a spanning tree to a base station where the aggregate functions are applied, the query compiler generates code that computes *partial aggregates*, which represent the intermediate state of an aggregate as data flows up the spanning tree. For example, the aggregate function `AVG` maintains a partial aggregate consisting of a count and a sum. These partial aggregates can be merged by adding their components pairwise, and local temperature readings can be added to an existing partial aggregate by adding the temperature reading to the sum and incrementing the count. When the partial aggregate reaches the base station, it is *reduced*; in the case of the `AVG` aggregate, this involves dividing the sum by the count.

Given a query, the compiler builds a Flask signal that carries partial aggregates received over the radio, one for each aggregate. The compiler also constructs a signal that produces tuples containing the local values of the selected attributes, clocked at the specified period. Partial aggregates from other nodes are merged and kept as local state until a value is received via the local attribute signal. When this occurs, the local attributes are merged into the existing partial aggregate which is forwarded up the spanning tree, and the local partial aggregate state is reset.

The types of these signals vary from query to query, so the query compiler uses existential types in its representation of intermediate values produced during compilation. We also use the trick of representing heterogeneous lists, which are necessary for representing query results, with nested tuples. The full query compiler is less than 200 lines of Haskell, not counting the parser. Our example query produces a residual program consisting of over 1000 lines of nesC.

7. Evaluation

Our goal in evaluating Flask is to demonstrate that it can be used to write real sensor network applications while achieving acceptable overheads (in terms of CPU and memory) compared to more conventional approaches. We achieve this through a number of microbenchmarks that quantify the overhead imposed by Flask and by showing data from a simple query compiled with the compiler developed in Section 6.2 and run on a testbed of TelosB motes.

7.1 Microbenchmarks

Figure 5 shows CPU cycle counts for several microbenchmarks that typify tasks performed by sensor network applications. For each benchmarked task, both a version written in Flask and a version written by hand in NesC were measured. For the Flask version, we measured both an unoptimized residual program and a version for which Flask performed a small optimization when generating code, which we describe momentarily. All measurements were taken using Avrora (Titzer et al. 2005), a cycle accurate simulator for MicaZ motes[1]. We chose to benchmark MicaZ binaries instead of TelosB binaries because we are not aware of a mature, cycle-accurate simulator for the TelosB platform.

The "Filter" benchmark applies a filter to a synthetic signal, only passing values that are greater than zero. The "Chained maps" function applies a signal function, formed by composing an `increment` signal function with itself ten times, to a synthetic signal. The NesC version wires together ten instances of an `Increment` component instead of using function composition. The "Windowed average" benchmark outputs the average of the previous ten values taken from an input signal. The "EWMA" and "Eruption detection" benchmarks are as described earlier in the paper.

[1] In contrast to TelosB motes, MicaZ motes have an 8-bit CPU and 8K of RAM instead of a 16-bit CPU and 10K of RAM

	TinyOS Base		Communications		Common		Application		**Total**	
	ROM	RAM	ROM	RAM	ROM	RAM	ROM	RAM	ROM	RAM
NesC	8360	1070	2840	1638	10360	940	16380	5540	37940	9188
Flask	8308	926	7498	2657	10506	916	12522	5168	38834	9667

Figure 6: **MSP430 binary size (in bytes) for the NesC and Flask implementations of the volcano monitoring application.**

	Flask	Flask (opt.)	NesC
Filter	4	4	5
Chained maps	2	2	6
Windowed average	1076 ± 134	1076 ± 134	348 ± 133
EWMA	1110 ± 43	1015 ± 43	799 ± 59
Eruption detection	2423 ± 7	2008 ± 7	1519 ± 7

Figure 5: **Microbenchmark CPU cycle counts on a simulated MicaZ mote.** *Values are shown over a one-minute simulated run; standard deviations are shown where significant.*

In the latter three benchmarks, Flask imposes a significant overhead in CPU cycle count. With optimization enabled, this overhead is in most cases reduced, although not eliminated. The optimization performed is just recursive argument flattening—when generating code for a function that takes a tuple as an argument, instead of passing the tuple as a single value, it is unboxed and its constituent parts passed as individual arguments. It turns out that first two microbenchmarks use only values of base type—integers and doubles—whereas the latter three microbenchmarks use tuples and/or algebraic data types. In the cases where only base types are involved in computations, Flask imposes no computational overhead. We therefore speculate that the overhead imposed by Flask is almost entirely due to the wrapping and unwrapping of these values; this claim is supported by the fact that simply flattening function arguments removes a substantial amount of overhead in the EWMA benchmark, which involves a substantial amount of wrapping and unwrapping. We believe that inlining combined with case elimination,as described in (Jones and Marlow 2002), would serve to eliminate most of the additional overhead.

To evaluate memory overhead, we compare the Flask and NesC versions of the complete volcano monitoring system described in Section 5. Figure 6 shows a breakdown of RAM and ROM sizes for the MSP430 binary. We break the code down by TinyOS core components, communications (including routing and command dissemination), common modules (sampling and flash storage), and application-specific code. As the figure shows, the Flask version uses 2% more ROM and 5% more RAM than the original NesC code despite the increased complexity of the Flows routing layer. The NesC application code is more complex due to increased control logic implemented more efficiently in Flask. We were careful to size static data structures (such as message buffers) equivalently in both systems.

7.2 Running Flask on real hardware

Our final evaluation of Flask demonstrates a FlaskDB query that reports the average temperature across a sensor network every 10 seconds. The goal of this evaluation is to show that Flask works on real hardware. Our example query was run on Motelab (Werner-Allen et al. 2005b), consisting of 160 TelosB motes spread over three floors of a building. Query results were delivered to a single root node using a spanning tree. The root node was in a central location on the second floor. During our trial run, only 157 of the 160 nodes were able to deliver at least one packet to the base station over a multi-hop route.

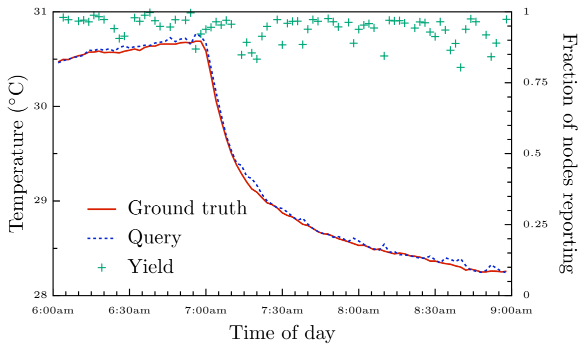

Figure 7: **Ground truth and query results for the FlaskDB query** `SELECT COUNT(id), AVG(temp) INTERVAL 10s`**.**

Figure 7.2 shows the average temperature computed by the query over a 3 hour run. We also show the ground truth value of the average temperature, obtained by having each node report its current temperature to its serial port, which is logged to an external database by the testbed server. The temperature increases as the sun heats the building in the early morning, and then sharply decreases as the air conditioning kicks in at 7 a.m. Note that deviation from ground truth is correlated to poor yield—we speculate that the distant, less well-connected nodes that have trouble delivering data to the centrally located base station are in warmer parts of the building.

The yield of the query, measured as the fraction of all 160 nodes that reported values to the base station, is plotted on the secondary y axis. Note that this includes the 3 nodes that could never find a path to the base station. Despite the fact that the deployment is spread over three full floors of a building, yield is still always above 80%.

8. Related Work

Clearly we took inspiration from the existing work on functional reactive programming, which has been used to build reactive systems in many domains, including robotics (Peterson et al. 1999; Pembeci et al. 2002; Hudak et al. 2003), animation (Elliott and Hudak 1997), and graphical user interfaces (Courtney and Elliott 2001). Our work is motivated by many of the same goals, namely exploiting the advantages of high-level languages to build reactive systems. However, unlike traditional FRP systems, we must cope with severely resource constrained target platforms and provide integration with existing code.

Closer to our work is the work on Real-Time and Event-Driven FRP (Wan et al. 2001a,b; Wan 2002). These systems seek to provide static time and space bounds on FRP systems with the goal of being able to run FRP programs on embedded devices. Unlike Real-Time FRP, we cannot make the assumption that all signals are synchronously driven by a global clock. Event-Driven FRP eliminates this assumption, but the resulting language is so pared down that it loses most of the appeal FRP holds in the first place. Our work seeks to constrain the space and time behavior of node-level code while maintaining as much of the power of FRP as possible. We have also implemented an entire, working programming platform that runs on severely resource-constrained devices. Real-Time FRP focuses less on implementation and to our knowledge the only program running on a device in a class comparable to our target platform is a simple robot movement controller, which is itself a slave to an FRP program running on a PC-class device.

Also related to our work are the many programming environments for sensor networks EnviroSuite (Lu et al. 2005), Semantic Streams (Whitehouse et al. 2006), Kairos (Gummadi et al. 2005), Regiment (Newton et al. 2007), and Abstract Task Graphs (Bakshi et al. 2005a). These systems offer a range of programming models at different levels of abstraction and are often tailored for a fairly narrow range of target applications. For example, EnviroSuite (Lu et al. 2005) is targeted at tracking applications, while Semantic Streams (Whitehouse et al. 2006) provides a logic-based language for composing distributed data-processing services.

The system that is most similar to Flask is WaveScript (Newton et al. 2008), a general-purpose stream processing language which targets more powerful devices than Flask. Like Flask and unlike StreamIT (Gordon et al. 2002), WaveScript focuses on asynchronous data streams. In contrast to WaveScript, we make a careful distinction between the meta and object level. The WaveScript developers eschew the syntactic distinction between meta and object level terms, claiming that the extra complexity inherent in explicit program staging imposes a "cognitive burden" on the programmer. Instead of explicit staging, WaveScript relies on partial evaluation to produce a residual program. The down side of this approach is that even if there exists some partial evaluation strategy that would allow a given program to be compiled efficiently, there is no guarantee that the WaveScript compiler implements this strategy. In the end, both Flask and WaveScript require that programmers reason about program staging. We believe that being upfront and explicit about this staging imposes less of a burden on the programmer than does requiring him to reason about the inner-workings of a compiler.

9. Conclusions and Future Work

We have described Flask, a programming environment that brings FRP-style programming to sensor networks. By carefully staging computations, Flask constrains the space and time behavior of node-level code while allowing the programmer to use the power of functional programming to construct sensor network programs. We have shown that Flask is flexible enough to build systems that run on real sensor network hardware and that it has acceptable overhead. Although targeted at sensor networks, we believe the techniques Flask uses to embed object languages in Haskell are applicable to a wide range of domains.

We envision many interesting future directions for Flask. As a first step, we hope to improve the Flask compiler to remove the overhead incurred by the use of tuples and algebraic data types. We also plan to push Flask by using it to build more real-world applications. Finally, Flask raises many interesting issue related to metaprogramming which we hope to explore.

Acknowledgments

This research is supported by the National Science Foundation under grant number CNS-0519675. We would also like to thank Geoff Werner-Allen for providing technical support for the MoteLab testbed.

References

Andreas M. Ali, Kung Yao, Travis C. Collier, Charles E. Taylor, Daniel T. Blumstein, and Lewis Girod. An empirical study of collaborative acoustic source localization. In *IPSN '07*, pages 41–50, 2007.

Amol Bakshi, Viktor K. Prasanna, Jim Reich, and Daniel Larner. The abstract task graph: A methodology for architecture-independent programming of networked sensor systems. In *Proc. Workshop on End-to-End, Sense-and-Respond Systems, Applications, and Services*, pages 19–24, 2005a.

Amol Bakshi, Viktor K. Prasanna, Jim Reich, and Daniel Larner. The abstract task graph: a methodology for architecture-independent programming of networked sensor systems. In *EESR '05: Proceedings of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services*, pages 19–24, Berkeley, CA, USA, 2005b. USENIX Association.

Elaine Cheong, Judith Liebman, Jie Liu, and Feng Zhao. TinyGALS: A programming model for event-driven embedded systems. In *SAC*, pages 698–704, 2003.

Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM.

Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *2001 Haskell Workshop*, September 2001.

Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, volume 32(8), pages 263–273. ACM, August 1997.

David Gay, Philip Levis, J. Robert von Behren, Matt Welsh, Eric A. Brewer, and David E. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. Programming Language Design and Implementation (PLDI '03)*, pages 1–11. ACM, 2003.

Omprakash Gnawali, Ki-Young Jang, Jeongyeup Paek, Marcos Vieira, Ramesh Govindan, Ben Greenstein, August Joki, Deborah Estrin, and Eddie Kohler. The Tenet architecture for tiered sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 153–166, New York, NY, USA, 2006. ACM Press.

Michael Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Christopher Leger, Andrew A. Lamb, Jeremy Wong, Henry Hoffman, David Z. Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *ASPLOS '02*, 2002.

E. Goto. Monocopy and associative algorithms in an extended LISP. Technical Report 74-03, Univ. of Tokyo, Information Science Lab., May 1974.

Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. Macro-programming wireless sensor networks using Kairos. In *Proc. DCOSS'05*, 2005.

Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *ASPLOS '00*, pages 93–104, 2000.

Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.

John Hughes. Generalising monads to arrows. *Sci. Comput. Program*, 37 (1-3):67–111, 2000.

John Hughes. Programming with arrows. In *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 73–129. Springer, 2004.

Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

Simon L. Peyton Jones and Simon Marlow. Secrets of the glasgow haskell compiler inliner. *J. Funct. Program*, 12(4&5):393–433, 2002.

Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP '06*, pages 50–61. ACM, 2006.

Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering (functional pearl). *ACM SIGPLAN Notices*, 35(9):280–292, 2000.

Nupur Kothari, Ramakrishna Gummadi, Todd Millstein, and Ramesh Govindan. Reliable and efficient programming abstractions for wireless sensor networks. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI '07)*, pages 200–210, 2007.

Philip Levis, David Gay, and David Culler. Active sensor networks. In *NSDI '05: Proceedings of the Second USENIX/ACM Symposium on Networked System Design and Implementation*, 2005.

Liqian Lu, Tian He, Tarek Abdelzaher, and John Stankovic. Design and comparison of lightweight group management strategies in EnviroSuite. In *Proc. International Conference on Distributed Computing in Sensor Networks (DCOSS)*, Marina Del Rey, CA, June 2005.

Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems*, 30(1):122–173, 2005.

Geoffrey Mainland. Why it's nice to be quoted: quasiquoting for haskell. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 73–82, New York, NY, USA, 2007. ACM.

Ryan Newton, Greg Morrisett, and Matt Welsh. The regiment macroprogramming system. In *Proc. IPSN '07*, 2007.

Ryan Newton, Lewis Girod, Michael Craig, Sam Madden, and Greg Morrisett. Wavescript: A case-study in applying a distributed stream-processing language. Technical Report MIT-CSAIL-TR-2008-005, MIT CSAIL, 2008.

Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, October 2002. ACM Press.

Izzet Pembeci, Henrik Nilsson, and Greogory Hager. Functional reactive robotics: An exercise in principled integration of domain-specific languages. In *PPDP '02*, October 2002.

John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. *Lecture Notes in Computer Science*, 1551: 91–105, 1999.

Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.

Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. Eon: a language and runtime system for perpetual systems. In *Proc. SenSys '07*, pages 161–174, New York, NY, USA, 2007. ACM.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66, 2007.

Walid Taha. Multi-stage programming: Its theory and applications. Technical Report CSE-99-TH-002, Oregon Graduate Institute of Science and Technology, 1999.

Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1–2):211–242, 2000.

Ben Titzer, Daniel K. Lee, and Jens Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Fourth International Symposium on Information Processing in Sensor Networks (IPSN '05)*, pages 477–482, 2005.

Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *International Conference on Functional Programming (ICFP '01)*, Florence, Italy, September 2001a.

Zhanyong Wan. *Functional Reactive Programming for Real-Time Reactive Systems*. Ph.d. dissertation, Computer Science Department, Yale University, October 2002.

Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven FRP. *Lecture Notes in Computer Science*, 2257:155+, 2001b.

Stephen Weeks, Matthew Fluet, Henry Cejtin, and Suresh Jagannathan. `http://www.mlton.org/`.

Geoff Werner-Allen, Jeff Johnson, Mario Ruiz, Jonathan Lees, and Matt Welsh. Monitoring volcanic eruptions with a wireless sensor network. In *Proc. Second European Workshop on Wireless Sensor Networks (EWSN '05)*, January 2005a.

Geoff Werner-Allen, Pat Swieskowski, and Matt Welsh. Motelab: A wireless sensor network testbed. In *Proc. Fourth International Conference on Information Processing in Sensor Networks (IPSN'05), Special Track on Platform Tools and Design Methods for Network Embedded Sensors (SPOTS)*, April 2005b.

Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proc. 7th USENIX OSDI*, Seattle, WA, Nov 2006.

Kamin Whitehouse, Feng Zhao, and Jie Liu. Semantic Streams: a framework for declarative queries and automatic data interpretation. Technical Report MSR-TR-2005-45, Microsoft Research, One Microsoft Way, Redmond, WA 98052, April 2005.

Kamin Whitehouse, Jie Liu, and Feng Zhao. Semantic Streams: a framework for composable inference over sensor data. In *Proc. Third European Workshop on Wireless Sensor Networks (EWSN)*, Zurich, Switzerland, February 2006.

Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. *POPL '03*, 2003.

Ning Xu, Sumit Rangwala, Krishna Kant Chintalapudi, Deepak Ganesan, Alan Broad, Ramesh Govindan, and Deborah Estrin. A wireless sensor network for structural monitoring. In *SenSys '04*, pages 13–24, 2004.