Compiling Functional Reactive Macroprograms for Sensor Networks

by

Ryan Rhodes Newton

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

[February 2005]
~~January 2005~~

© Massachusetts Institute of Technology 2005. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
January 31, 2005

Certified by . . . . [illegible] . . . .
Arvind
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . . [illegible] . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
MAR 14 2005
LIBRARIES

ARCHIVES

Compiling Functional Reactive Macroprograms for Sensor Networks

by

Ryan Rhodes Newton

Submitted to the Department of Electrical Engineering and Computer Science
on January 31, 2005, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

Sensor networks present a number of novel programming challenges for application developers. Their inherent limitations of computational power, communication bandwidth, and energy demand new approaches to programming that shield the developer from low-level details of resource management, concurrency, and in-network processing. To answer this challenge, this thesis presents a functional macroprogramming language called Regiment. The essential data model in Regiment is based on regions, which represent spatially distributed, time-varying collections of state. The programmer uses regions to define and manipulate dynamic sets of sensor nodes . A first compiler for Regiment has been constructed, which implements the essential core of the langugae on the TinyOS platform. This thesis presents the compiler as well as an intermediate language developed to serve as a Regiment compilation target.

Thesis Supervisor: Arvind
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank Professor Arvind for teaching me how to sell systems for *building* systems; Matt Welsh for his macroprogramming perspective and his hand in Regiment's design; and Gerald Sussman for introducing me to Amorphous Computing. I would also like to thank Nirav Dave, Michael Pellauer, Greg Morisett and Norman Ramsey for their enlightening discussion on languages and functional programming.

I would especially like to thank my wife for her unwaivering support, including reading drafts. Also I must thank our children, Aspen and Nimbus, for their unwaivering ... purring. And, of course, I thank my parents for being supportive enough that I can take them for granted.

Contents

List of Figures

Chapter 1

Introduction

A sensor network represents a complex, volatile, resource-constrained cloud of sensors capable of collaborative sensing and computing. Programming such an entity requires new approaches to managing energy usage, performing distributed computation, and realizing robust behavior despite message and node loss.

One approach is to program the sensor network as a whole, rather than writing low-level software to drive individual nodes. Not only does such an approach raise the level of abstraction for developing novel programs, we argue that the only way to address the complexity of the underlying substrate is through automatic compilation from a high-level language. Today, few computer scientists would doubt the value of high-level languages for programming individual computers, or even groups of machines connected in a traditional network. We wish to take this approach to the next level and provide a *macroprogramming* environment for a network of sensors that automates the process of decomposing global programs into complex local behaviors.

This thesis presents a functional macroprogramming language for sensor networks, called *Regiment*. The essential data model in Regiment is based on *regions*, which represent spatially distributed, time-varying collections of node state. The programmer uses these to express interest in a group of nodes with some geographic, logical, or topological relationship, such as all nodes within k radio hops of some *anchor* node. The corresponding region represents the set of sensor values across the nodes in question. The operations permitted on regions include *fold*, which aggregates values across nodes in the region to a particular anchor, and *map*, which applies a function over all values within a single region. Operationally, map requires no communication between elements, whereas fold requires the collapse of data to a single physical point.

Regiment is a *purely functional language* that does not permit input, output, or direct manipulation of program state. Regiment uses monads [32] to indirectly deal time-varying values such as sensor reading. As in other functional language designs, this approach gives the compiler considerable leeway in terms of realizing region operations across sensor nodes and exploiting redundancy within the network. The Regiment compiler transforms a network-wide macroprogram into an efficient nodal program expressed as a *Token Machine.* A program in the Token Machine Language (TML) is a simple state machine that generates and processes named tokens, communicating only with neighbors in the network.

Contributions

The contributions of this thesis are: the design of the Regiment language, the implementation of the first Regiment compiler, and the implementation of the Token Machine run-time with an associated simulator. The Regiment language takes ideas from Functional Reactive Programming [17] to a new level, and its implementation puts a stake in the ground for sensor network macroprogramming.

The compiler is written in Scheme and the Token Machine assembler in Haskell. Together, they produce output code in a C-variant called NesC, which executes on the TinyOS platform. The current prototype supports a core subset of the full Regiment language.

The Token Machine intermediate language and run-time environment forms the basis for running Regiment programs on TinyOS motes (such as the Mica2 and MicaZ). But it also provides a potential compilation target for other high-level languages. Its primary benefits derive from its simple memory model and uniform atomic action model of concurrency. Because human readable/writable, it is also a low-overhead way to write simple sensor network applications.

Structure

Chapter two describes the Regiment language, including the aspects of its specification that have not been fully implemented in the current compiler. We showcase its operators and idioms, present example programs, and provide a non-formal description of Regiment's semantics. Chapter three describes the Token Machine Language (TML) and the abstract model model its based on — the Distributed Token Machine (DTM). The stage is then set for chapter four to present the existing Regiment compiler and simulator. If desired, it would be possible to read chapter three on-demand as it is referenced by chapter four. In chapter four the conceptual mapping between high-level Regiment programs and low-level TML programs is discussed. Then, the passes that comprise the Regiment compiler are described individually and we present the compiler's multi-step simulation framework (multiple levels of detail) which allows executable intermediate code throughout the entire compilation. Finally, chapter five concludes.

Chapter 2

The Regiment Programming Language

2.1 Related Work

For sensor networks, progress in macroprogramming has largely been domain specific. We have seen: languages for global-to-local compilation of spatial pattern formation [35, 28, 14]; Envirotrack [2], which exposes tracked objects as language *objects* (analogous to the way we expose regions); and, of course, database systems for querying sensor data [50, 31].

Amorphous Computing

The Amorphous Computing research effort has pursued the broad goal of engineering aggregate behaviors for dense ad-hoc networks (paintable computers, Turing substrates). Their work focuses on pattern formation, taking inspiration from developmental biology. They demonstrate how to form coordinate systems [34], arbitrary two and three dimensional shapes [28], arbitrary graphs of "wires" [14], and oragami-like folding patterns [35]. Yet the Amorphous Computing effort has not to date provided a model for *programming* rather than *pattern formation.* In addition, the target platforms envisioned by the Amorphous Computing effort differ significantly from existing wireless sensor networks.

Database approaches

The database community has long taken the view that declarative programming through a query language provides the right level of abstraction for accessing, filtering, and processing relational data. Recently, query languages have been applied to sensor networks, including TinyDB [31], Cougar [50], and IrisNet [36]. While these systems provide a valuable interface for efficient data collection, they do not focus on providing general-purpose distributed computation within a sensor network. For example, it is cumbersome to implement arbitrary aggregation and filtering operators and arbitrary communication patterns using such a query language. We argue that a more general language is required to fully realize the potential for global network programming.

There has also been a body of work on extending programming languages to deal with database access: database programming languages or DBPLs. Many types of languages have been used in this work, including functional ones. Functional DBPLs include FAD [8] and IPL [4]. Regiment differs from these languages in being explicitely concerned with: distributed processing, spatial processing, streaming data, and with the volatility of its substrate—sensor networks.

Stream processing languages

Stream processing is an old subject in the study of programming languages. Functional Reactive Programming (FRP) is a recent formulation which uses modern programming language technology (including monads [32] and type classes [46]) to allow purely functional languages to be able to deal comfortably with real time events and time-varying streams. FRP is the inspiration for Regiment's basic type system.

Regiment's problem domain also overlaps with recent work in extending databases to deal with continuous queries over streaming data, such as STREAM [5], Aurora [51], Medusa [51], and recently Borealis [1]. Regiment aims to utilize many optimization techniques developed in this body of work, but at the same time Regiment occupies a different niche. Regiment runs on tiny sensor nodes, utilizes local communication only, and operates in a scenario where data sources are numerous and not statically known — for example all nodes whose light readings exceed a threshold.

2.2 The functional macroprogramming approach

The traditional method of programming sensor networks is to write a low-level program that is compiled and installed in each individual sensor. This amounts to a programming model consisting of access to sensor data on the local node, coupled with a message-passing interface to radio neighbors. In contrast, our macroprogramming model captures the entirety of the sensor network state as a global data structure. The changing state of each sensor originates a stream of data at some point in space. Collectively they form a global data structure. As an intuition for why this is the right approach, consider that matrix multiply algorithms are far simpler to state in terms of matrices and vectors than as parallel programs implemented in MPI.

To express sensing and communication within local groups of nodes, regions encapsulate subsets of the global network state that can be manipulated by the programmer as single units. They represent the time-varying state of a time-varying group of nodes with some geographic or topological relationship. Communication patterns for data sharing and aggregation can be efficiently implemented within such local regions [47, 48].

2.2.1 Why a functional language?

We propose that functional languages are intrinsically more compatible with distributed implementation over volatile substrates than are imperative languages. Prominent (call-by-value) functional languages include Lisp, Scheme and OCaml. Functional languages have been used to explore high-level programming for parallel machines—such as NESL [10] and *LISP [43]—and for distributed machines [38]. In our system, we get the most benefit from restricting ourselves to a *purely* functional (effect free), call-by-need language similar to Haskell [26].

Purely functional languages essentially hide the direct manipulation of program state from the programmer. In particular, the program cannot directly modify the value of variables; rather, all operations must be represented as *functions.* Monads [32] allow mutable state to be represented in a purely functional form. For sensor network applications, abstracting away the manipulation of state allows the compiler to determine how and where program state is stored on the volatile mesh of sensor nodes. For example, to store a piece of data reliably, it may be necessary to replicate it across multiple nodes in some consistent fashion. Using a functional language makes consistency moot; immutable values can be freely replicated and cached.

Because functions are deterministic and produce no output, computation can be readily migrated or replicated without affecting program semantics. Another way to state this is that functional programs support *equational reasoning.* Program optimization in such a framework can be cast as semantics-preserving application of general program transformations [37].

Regions and streams (which contain time-varying data but not spatial extent) are encapsulated using monads. Monads are a programming language technology that enables the clear separation of semantics of different kinds of computation. One can cleanly embed foreign computations by revealing them as first class monadic values within a host language. These foreign computations then can be manipulated only through the monadic operators. Thus streams and regions may passed to and from functions, stored in variables, but the only access to the data inside them is through the map, fold, filter, and other operations described below.

Regiment has a host of algebraic properties which can be used together with a static cost model or dynamic profiling information to optimize performance and resource usage.

Another advantage of the functional programs is that it is straightforward to extract parallelism from their manipulation of data. For example, a function that combines data streams from multiple sensors can be compiled into a form that efficiently aggregates each data stream within the network. In addition to such *data parallel* operations, functional programs are implicitly parallel in their evaluation of function arguments [6]. The compiler can automatically extract this parallelism and implement it in a variety of ways, distributing operations across different sensor nodes.

2.3 The Regiment language

The goal of Regiment is to write complex sensor network applications with just a few lines of code. In this section we describe the Regiment language through several examples. A common application driver for complex coordination within sensor networks is that of tracking moving vehicles through a field of sensors each equipped with a proximity sensor of some kind (e.g., a magnetometer). We start by showing a simple Regiment program that returns a series of locations tracking a single vehicle moving through such a network.

```
let aboveThresh (p,x) = p > threshold
    read node =
       (read_sensor PROXIMITY node,
        get_location node)
in centroid (afilter aboveThresh
                     (amap read world))
```

We use a syntax similar to Haskell. Function applications are written as *f x y*; for example, `amap read world` represents the application of the *amap* function with two arguments: *read* and *world*. One important characteristic of functional languages is that they allow functions to be passed as arguments. Here, *amap* takes the function *read* as argument, and applies it to every value of the region *world*; we will discuss the details shortly. `afilter` filters out elements from a region that do not match a given predicate, in this case the *aboveThresh* function. And `centroid` is a function that computes the spatial center of mass of a set of sensor readings (where each reading is a scalar value coupled with the *(x,y)* location of the sensor that generated the reading). We assume that every node has access to an approximation of its Euclidean location in real space, though this assumption is not essential to the Regiment language.

So, this program can be interpreted as follows: a region is created that represents the value of the proximity sensor on every node in the network; each value is also annotated with the location of the corresponding sensor. Data items that fall below a certain `threshold` are filtered out. Finally, the spatial centroid of the remaining collection of sensor values is computed to determine the approximate location of the object that generated the readings.

2.3.1 Fundamentals: space and time

Regiment is founded on three abstract polymorphic data types. Polymorphic types are also called *generics*, and are similar in use to C++ templates; they enable generic data structures to be specialized for use with any particular type of data element. Below, the α argument to each type constructor signifies the particular type that it is specialized to hold.

- Stream α — represents a value of type α that changes continuously over time
- Space α — represents a physical space with values of type α suspended in it
- Event α — represents a discrete event that occurs at a particular point in time and that carries a value α when it occurs

The notion of Streams and Events is based on Functional Reactive Programming [17]. In this model, programs operate on a set of time-varying signals. A signal can change its behavior on the arrival of an event. In Regiment, signals become Streams and are used to represent changing sensor state or network status, Spaces represent the physical distribution of information across a network, and Events notify the program of meaningful changes to Streams, allowing triggers.

Because Regiment is a purely functional language, the Stream, Space, and Event types all describe first-class immutable values. This means that values of these types can themselves be passed as arguments, returned from functions, and combined in various ways. Semantically, we can think of each of the three types as having the following meanings:

- Stream $\alpha \approx$ Time $\rightarrow \alpha$
- Space $\alpha \approx$ Location $\rightarrow$ MultiSet α
- Event $\alpha \approx$ (Time , α)

That is, Streams may be formalized as abstract functions that map a time to the value at that time. This is not to say that we would ever *implement* a Stream object as such. Similarly, Spaces may be formalized as functions mapping a location to a set of values existing at that location. Events simply become tuples containing values paired with the associated time of their occurrence.

2.3.2 Areas, Regions, and Anchors

Now we formalize our region notion by introducing Regiment's Area and Region types. An Area is a generic data structure for representing volatile, distributed collections of data. A Region is a specific kind of Area used to represent the state of the real, physical network.

We saw before that a Space represents a "snapshot" of values distributed in space at a particular point in time. But we would like for those values—as well as the membership of values in that space—to change over time. To accomplish this we introduce the concept of an *Area*. If we visualize a `Space Int` as a volume with integers suspended throughout, then an `Area Int` would be an animated version of the same thing. The Area data type is built by using Stream and Space together:

$$\text{Area } \alpha = \text{Stream (Space } \alpha)$$

Note that, with this type, an Area's membership and physical extent may change over time. In fact, this type would allow the Area to become an entirely different Space at each point in time. (But the instability would cripple our implementation.) On the other hand, if Area were defined as a Space of Streams rather than Stream of Spaces, then its membership and spatial extent would be fixed but its values varying. Instead, both vary.

Areas are useful constructs, but they don't by themselves provide an initial foothold into the real world. How do we make that first Area? In order to refer to the state of specific sets of nodes in the real world, we define a *Region*, which is an *Area* of *Nodes*. A Node, in turn, is a datatype representing the state of a sensor node in the network at some point in time. It allows access to the node's state, such as its sensor readings, real world location, and the set of other nodes that are part of its communication neighborhood. The precise definition of the Node type, along with its basic operations, are shown in Figure 2-1.

A Region is created as a group of nodes with some relationship to one another such as"all nodes within k radio hops of node N," or "all nodes within a circle of radius r around position X." Regions may be formed in arbitrarily complex ways: using spatial coordinates, network topology, or by arbitrary predicates applied to individual nodes. Hence, Regions may be non-contiguous in space, and their membership may vary over time. The goal of a Region is to get a handle on a group of sensor nodes of interest for the purpose of localizing sensing, computation, and communication within the network. The special region `world` represents all nodes in the network.

One can form a Region by identifying a particular node that acts as the reference point for determining membership in the region: an *Anchor*. The Anchor also acts as the "leader" for aggregate operations in a Region, such as combining values from multiple sensors. Note that the specific node that fulfills the role of Anchor may change over time, for example, if a node fails or loses connectivity to others in the Region. Regiment guarantees that the Anchor object persists across node failures, which may require periodic leader elections. Anchors are useful for more implementing distributed algorithms that require marking reference points. For example, the Anchor Free Localization [39] algorithm begins by electing a number of anchors at the extreme "corners" of the network (e.g. first pick a random node, then pick the node furthest from that node, then the furthest from that). This takes a straightforward form in Regiment but is difficult to reconcile with a data-gathering, query-processing model as seen in TinyDB.

Examples of Regiment code for forming various Regions:

- **radio_neighborhood hops anch**:

  Forms a Region consisting of all nodes within **hops** radio hops of the given anchor.

- **circle radius anch**:

  Forms a Region consisting of all nodes whose geographical coordinates are within **radius** of **anch**.

- **knearest k anch**:

  Forms a Region consisting of the **k** nodes that are nearest **anch**.

2.3.3 Basic operations

Regiment defines a number of basic operations on Streams and Areas.

smap *f stream*

amap *f area*

smap applies a function *f* to every data sample within a Stream (across time), returning a new Stream. Similarly, *amap* applies a function *f* across every datum in the Area (across space and time).

afold *f init area*

An Area fold, or *afold*, is used to aggregate the samples from each location in the Area to a single value. The function *f* is used to combine the values in the Area, with an initial value of *init* used to seed the aggregation. *afold* returns a new Stream representing the aggregated values of the Area over time. For example, `afold (+) 0 area` generates a Stream of the time-varying sum of all values in *area*.

afilter *p area*

An Area filter, or *afilter*, pares down the elements of *area* to only those satisfying the predicate function *p*. This filtration must be updates dynamically as the values in *area* change over time.

Regiment also has operations for defining and handling events:

when *p stream*

whenAny *p area*

whenPercent *per p area*

when constructs an Event which fires when the current value of a stream satisfies the predicate *p*. *whenAny*, on the other hand, constructs an Event that fires whenever any single node in an Area matches a predicate *p*. *whenPercent* is similar to *whenAny* but the Event returned only fires when above a certain percentage of elements in the area meet the criteria—potentially an expensive (and difficult to implement) operation.

Using Events, two Streams can be sequenced into a single Stream using the *until* function:

until *event startstream handler*

until switches between Streams. The above call to *until* will produce values from *startstream* until such a time as *event* occurs. At that point, the *handler* (a function) is called on the value attached to the Event occurrence. This handler function must return a new Stream, that takes over producing values where *startstream* left off.

2.3.4 Spatial operations

Along with these basic operators, Regiment provides several explicitly spatial operations on Areas. For example:

- **sparsify** *percent area*:

  Make *area* more sparse. Each value in the Area flips a biased coin, and stays in the Area with the given probability. This randomization is only done the first time a value enters the Area. The sparse Area is not chaotically recomputed at every time step. **sparsify** can be used, for example, to "weed out" nodes from an overly dense Region.

- **cluster** *area*:

  Cluster a fragmented Area into multiple Areas, each of which is guaranteed to be spatially contiguous. The return type is an Area of Areas.

```
type Area a = Stream (Space a)
type Region = Area Node
type Anchor = Stream Node
   – Node: represents a physical mote in the context of a
   – communication network. Provides access to the node
   – state as well as the states of "neighbors".
type Node = (NodeState, [NodeState])

   – NodeState: all the relevent information for a
   – node: id, location, and a set of sensor values
   – (one for each sensor type supported by the node).
type NodeState = (Id, Location, [Sensor])
   – Sensor: force all sensor readings to be floats:
type Sensor = (SensorType, Float)
   – SensorType: predefined enumeration of sensor kinds.
type SensorType =
   PROXIMITY | LIGHT | TEMPERATURE ...

   – Function that returns the NodeState of a Node
get_nstate ::  Node -> NodeState
   – Returns the reading for a given SensorType. For
   – now we assume all nodes support all SensorTypes.
read_nstate ::
   SensorType -> NodeState -> Float

   – And here are two convenient short-hands:
   – Sensing function for Nodes
read_sensor typ nd =
   read_nstate typ (get_nstate nd)
   – Shorthand for reading location (via GPS, etc)
get_location nd =
   read_sensor LOCATION node
```

Figure 2-1: **Regiment's basic data types** (along with some helpful functions)

- **flatten** *area*:

  Flatten takes an Area of Areas and returns a single combined Area. This is the inverse of **cluster**.

- **border** *area*:

  Return a Region representing the set of nodes that form a boundary around the given *area*.

2.3.5 Example programs

Now we will return to our original example program and examine it in greater detail. Let us start by defining `centroid` using basic Regiment constructs.

```
  – This calcs a weighted avg of vectors.
  – Used to find center of sensor readings.
centroid area =
   divide (afold accum (0,0) area)

  – 'accum' produces a weighted sum.
  – 'wsum' - sum of weights.
  – 'xsum' - sum of scaled locations.
accum (wsum, xsum) (w,x) =
   (w + wsum, x*w + xsum)

  – 'divide' the stream of scaled location
  – values by the sum of the weights.
  – Backslash defines a function.
divide stream =
   smap (\(w,x) -> x/w) stream
```

The `centroid` function takes an area as an input and uses the `accum` function to fold that area down to a stream of sums of sensor readings paired with the scaled locations of each sensor in the region. The `divide` function divides the sum of scaled locations by the sum of the sensor readings. This effectively calculates the center of mass of the locations of those sensors, in a way that recomputes automatically over time.

Tracking multiple targets

Using the **cluster** operation, we can track the location of multiple targets, assuming that the set of nodes near a given target do not overlap:

```
 let aboveThresh (p,x) = p > threshold
     read node =
        (read_sensor PROXIMITY node,
        get_location node)
    selected = afilter aboveThresh
                       (amap read world)
    globs = cluster selected
in amap centroid globs
```

This program returns an Area providing approximate target locations for each target being tracked. Note that the number of targets in the Area will vary over time.

Resource efficiency with sentries

As a further refinement, consider a program that only initiates target tracking within the network if any of the nodes on the periphery of the network initially detect the presence of a target. This technique can be used to save energy on the interior nodes of the network, which only need to be activated once a target enters the boundary.

```
let aboveThresh (p,x) = p > threshold
    read node = (read_sensor PROXIMITY node,
                 get_coords node)
    selected  = afilter aboveThresh
                        (amap read world)
    targets   = amap centroid (cluster selected)
    sentries  = amap read (border world)
    event     = whenAny aboveThresh sentries
    handler ev = targets
in until event nullArea handler
```

The last line of the program initiates computation using the **until** primitive. Until `event` fires, the program returns an empty Area (**nullArea**). Once a target is detected by any of the `sentries`, the **nullArea** is supplanted by `targets`, the evaluation of which yields a stream of approximate target locations.

The reader might reasonably be worried that the above program produces a fragile implementation. If even one node in the sentry-border dies, might that let a target through? This depends on the quality of the implementation of the `border` operator.

A high quality implementation will respond to failures and have the border sealed again in a bounded amount of time. Also, the programmer may self-insure by making a two layer border as follows:

```
let sent1 = border world
    sent2 = border (subtract world sent1)
    thickborder = union sent1 sent2
    ...
```

Contour finding

The following program computes the *contour* between adjacent areas of the network. Sensor readings on one side of the contour are above a certain threshold, and readings on the other side are below. The contour is returned as a list of points lying along the contour.

```
let mesh = planarize world
    nodesAbove =
       afilter ((>= threshold) .
                (read_sensor SENSTYP))
               mesh
    midpoint nst1 nst2 =
       (read_nstate LOCATION nst1 +
        read_nstate LOCATION nst2) / 2
    contourpoints node =
       let neighborsBelow =
         filter ((< threshold) .
                 (read_nstate SENSTYP))
                (get_neighbors node)
       in map (midpoint (get_nstate node))
            neighborsBelow
    all_contourpoints =
       amap contourpoints nodesAbove
in
  afold append all_contourpoints
```

This program works by pruning the communication graph of the network into an approximately planar form. It then filters out a region of nodes—`abovethresh`—with `SENSTYP` reading above the threshold; this would be all the nodes to one side of the contour. The `contourpoints` function takes a node above the threshold and returns a list of midpoints between that node and each of its neighbors *below* the threshold (on the other side of the contour). Finally, *all_countourpoints* is aggregated by appending together all the individual lists of midpoints, thus yeilding the final countour-line—a Stream of lists of coordinates.

2.3.6 Feedback and exception handling

Because behavior of the sensor network is stochastic, the response from a region during any time period will involve only a subset of all the nodes that "should" be in that region. The programmer needs feedback on the quality of communication with the region in question. Thus the **fidelity** operator.

fidelity *area*

This operator returns a Stream representing the fidelity of an area as a number between zero and one (an approximation based on the number of nodes responding, spatial density, and estimated message loss).

The programmer will also want feedback about (and eventually control over) the *frequency* of a Stream.

get_frequency *stream*

allows the programmer to monitor the actual frequency of a Stream of values.

Thus, by using these two diagnostic streams, the programmer may set up "exception handlers". This is accomplished by constructing events which fire when fidelity or frequency falls out of the acceptable range. For example, if fidelity drops below a certain level, one may want to switch to a different algorithm. (However, it will be discussed in chapter 4, that the above operators are not yet fully implemented. The user can set frequency for streams, but not monitor frequency or fidelity.)

Chapter 3

Token Machine Intermediate Language

3.1 Introduction

In implementing Regiment, we want to build a layered abstraction. A number of ongoing projects aim to developing better programming tools and paradigms for sensor networks have resulted in monolithic software systems, which we wish to avoid. In order to develop high-level languages that compile into node-level programs, but to divide the complexity into multiple layers (like a network stack), it would be extremely valuable to define a common *intermediate language.*

We need an intermediate representation to abstract away the details of concurrency and communication while capturing enough detail to permit extensive optimizations by the compiler. Existing sensor network runtime environments, such as TinyOS, are too low-level to act as a desirable compilation target. This is especially true in Regiment. Because of its high level nature, the semantic gap between Regiment and NesC is large, making the task of compilation daunting.

In this chapter, we propose an intermediate language for sensor networks, called the Token Machine Language (TML). TML is based on a simple abstract machine model, which we call Distributed Token Machines (DTMs). Distributed Token Machines provide a simple execution and communication model based on *tokens.* Communication happens through token messages which are typed messages starting with a token containing a small payload. Tokens are associated with *token handlers* that are executed upon reception of a token message (either locally or from a radio message). The set of tokens that each node is holding is reflected through the DTM model, providing a form of distributed state management. Tokens are akin to Active Messages [45], although DTMs provide additional facilities for structuring execution, concurrency, state management, and communication that make this model more attractive as a compilation target for sensor network applications.

Our goal is to define an intermediate language for sensor network programming that:

1. Provides simple and versatile abstractions for communication, data dissemenation, and remote execution;

2. Constitutes a framework for network coordination that can be used to implement sophisticated algorithms such as leader election, group maintenance, and in-network aggregation; and,

3. Is high-level enough to serve as an effective compilation target

TML is meant to be *lightweight* in every respect. In terms of *performance*, TML must map efficiently onto the event-driven semantics of existing sensor network operating systems, such as TinyOS. In terms of *versatility*, TML must be primitive enough to construct a wide range of higher-level systems. At the same time, for *simplicity* TML must mask the complexities of the underlying OS and runtime environment. Simplicity and regularity are the keys to making TML an effective target language for compilers. These motivations differentiate TML from traditional intermediate languages, such as the Java Virtual Machine and CLI, which are primarily motivated by portability and safety. Instead, we are drawing on the lineage of systems such as the Threaded Abstract Machine (TAM) [15], which aims to provide an appropriate level of granularity to achieve abstraction without sacrificing performance or versatility.

With TML we wish to capture in-network coordination in a systematic and reusable way. TML provides a unified abstraction for communication, execution, and network state based on tokens. Specifically, all communication in TML is accomplished by dissemenation of tokens, which cause token handlers to execute on nodes that receive the token, as well as storage of the token in the node's memory. This approach allows an application to refer to the tokens as well as the set of nodes holding a particular token as semantically meaningful units. For example, tokens make it straightforward to implement network abstractions, such as *gradients*, which involve flooding a token throughout all or part of the network and constructing a spanning tree back to the origin of the gradient.This is a common routing model in sensor networks [24, 31] which can be implemented directly in TML in a manner that meshes directly with its token abstraction. A more advanced example would be the construction of a routing infrastructure that can route a given token to all the holders of another token.

The rest of this chapter is organized as follows. In Section 3.3, we present the semantics of the Distributed Token Machine model. The DTM defines an abstract machine, not a complete intermediate language. In Section 3.4, we show how to realize the DTM model as the Token Machine Language. In Section 3.5, we illustrate the use of TML for writing several simple applications, including a distributed event detector and a decentralized leader election algorithm. Finally, we will discuss our implementation of TML on top of TinyOS. We show that the TML abstraction introduces extremely little overhead in terms of code sizes, and only modest overhead in terms of RAM usage and execution speed.

3.2 Related Work

To our knowledge, however, the only direct attempt to provide an intermediate representation for compilation of sensor network programs has been the Maté virtual machine. The Maté engine interprets a compact bytecode language for sensor networks [29]. The focus is on energy-efficient dynamic reprogramming, application specific VM extensions, and safety. More recently, Maté has been extended to act as an intermediate language for several higher-level languages [49]. Our work shares this goal, however, Maté does not directly address the issue of high-level communication abstractions. Maté provides low-level access to radio communication, and has also been extended to support the abstract regions [47] communication model. TML extends upon the Maté approach by capturing a higher-level abstract machine that integrates a coordinated communication and execution model.

In addition to virtual languages, there are a range of other efforts to improve the infrastructure: from improvements to the NesC language, to reusable run-time services. The Sensor Network Application Construction Kit (SNACK) [20] makes a number of modularity improvements NesC, enabling more easily reusable components. This component composition language remedies a number of ills plaguing NesC developers, but does not have the clean and simple semantics desirable in a compiler backend. The Impala system [30] enables also application modularity and network reprogramming. MagnetOS [9] aims to provide automated assignment of Java components to parts of an ad-hoc network, but has not been scaled down to work on resource-constrained sensor networks.

Other middleware is more concerned with communication abstractions. Spatial Programming [12] uses Smart Messages to provide content-based spatial references to embedded resources. For example, the programmer may refer to the first available camera in a given (predefined) spatial region. The entity maintenance mechanism described in [11] enables tracked objects to be viewed as logical objects and serve as communication endpoints. A few communication models in particular had an influence on TML. Directed Diffusion [24] and SPIN [22] are paradigms for source-sink communications over named data (but are specific to that class of applications). It is from these projects — and from the work on gradients in Amorphous Computing [3] — that we are inspired to incorporate the general purpose gradient and aggregation interface of section 3.5.1.

TML is heavily inspired by Active Messages, which were originally conceived as a mechanism for efficient message passing in parallel architectures [45]. The original Active Messages work focused on integrating communication and execution for constructing parallel applications. Active Messages has found a new home in TinyOS [23], although it is used there primarily as a radio message format, rather than for introducing specific semantics for the execution of message handlers. For example, the TinyOS variant of AM does not specify how messages interact with the link and network layers of a protocol stack. In TML, the only communication abstraction provided is that of single-hop message broadcast, which can be used to build higher-level abstractions while staying within the token-oriented paradigm.

Ideally, TML could serve as an intermediate language for the compilation of high level programs other than Regiment programs — such as the SQL-like queries found in Cougar [50] and TinyDB [31]. While TinyDB compiles its queries into a lean query specification that is executed by individual nodes, the TinyDB system itself is monolithic and provides a wide range of functionalit — including spanning tree formation, query dissemenation, query optimization, and in-network aggregation. These features are not available to the TinyDB programming model except indirectly through SQL queries. Rather, by compiling TinyDB queries into TML, we could support a broader range of optimizations and tools that operate on the intermediate language level, allowing innovations to be shared across the language boundaries.

3.3 Distributed Token Machines

A Distributed Token Machine (DTM) is an abstract model for computation in an dynamic, asynchronous, ad-hoc network with failures and message loss. The DTM structures concurrency, communication and storage, but it makes few assumptions about what language is used to describe the computations inside each token handler. To able to use the architecture one must complete it by attaching a concrete language for handlers, which is embodied by the Token Machine Language.

3.3.1 Execution Model

In the DTM execution model, each node in the network holds some number of *tokens* at any point in time (as well as statically sized shared memory). Each of the stored tokens has an associated *token handler* which is executed upon receipt of the token, as described below, and a *memory* which is a set of private variables that may only

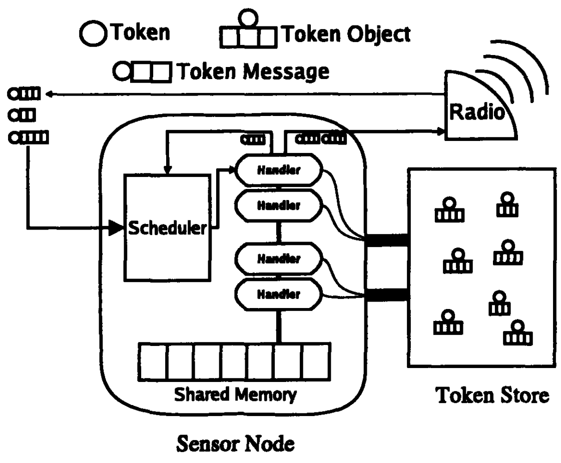


Figure 3-1: The structure of a node in the DTM model.

be accessed by that token handler. The stored token along with its private memory is referred to as a *token object.* In OOP terms, each token object can be thought of as an object with only private fields and only one method. Each handler executes atomically and in a statically bouded amount of time, which simplifies memory consistency issues and makes precise scheduling possible. If this seems restricted, it is because DTMs are *intended* to be highly restricted. Their restriction is a boon to compilers and analysis tools that work with the architecture. Their selling point is the ease with which expressive languages can be compiled down to them *in spite* of their simplicity.

They have the core ingredients necessary to build up higher level programming features. For, example, it would be straightforward to map multiple token "methods" down onto the one supported handler action. Similarly, issues of memory scope and protection can be compiler controlled. Most importantly of all, a compiler can break a long sequence of coordinated actions into a set of token handlers that interact in predictable ways.

Thus, a node in the network consists of heap, storing local token objects which we refer as the token store, as well as a scheduler that orders incoming token messages and some unspecified computing machinery able to execute handler functions (pictured in Figure 3-1). Token messages simply refer to tokens that travel over network channels along with their associated payloads. As in Active Messages, the token-name is at the start of each message. Once a message comes through the scheduler, the token name directs it to the corresponding handler. If the a coresponding token object is not already present in the token store, its memory is allocated and initialized (to zero) before the handler begins running. (While the message is still pending in the scheduler it consumes no memory in the token store.) The hanadler consumes the message payload and executes, possibly reading and modifying the token's private memory and node's shared memory in the process.

A complete DTM must have a fixed and finite set of tokens $\mathcal{T}$, with a fixed private memory size and bounded message payload size for each $\mathcal{T}_i$. DTMs allow *first class* tokens. Because there are only a finite number of tokens $\mathcal{T}_i$, a token name can be encoded as a fixed length sequence of bits. In the DTM model we allow tokens to be encoded as data and transfered in messages or stored in memories.

3.3.2 Handlers and their Communication Interface

Thusfar the only constraints we have put on the handlers is that they be computable by the processing machinery of the nodes in our network, that they execute atomically and only modify the appropriate memory. But executing handlers need to post new messages; they must interact with the scheduler and the token store. We still do not wish to specify exactly what language is used for describing handlers — we do not care, for example, what concrete syntas or data types are used — but we do wish to specify the interface through which the handler must interact with the rest of the system. It is as follows.

- schedule($\mathcal{T}_i, priority, data \ldots$)
- timed_schedule($\mathcal{T}_i, time, data \ldots$)

  The **schedule** operation inserts a token message in a nodes' own scheduler. *data*... is the payload of the token message. **timed_schedule** is a version of **schedule** which schedules a message to execute locally at a precise time—after a given number of milliseconds. This sets a hard requirement, overriding whatever strategy the scheduler is using to order messages. But in general the scheduler may use an implementation specific algorithm for scheduling incoming messages.

The acceptable values for the *priority* argument are also implementaton specific. The scheduler even has the ability to drop remotely received messages – because the DTM model is defined in terms of lossy channels. However, the scheduler is expected to follow certain rules. It must respect the *relative order* of all token messages produced by during the execution of a single atomic token handler. Thus two consecutive **schedule** commands in a handler will have their order preserved.

- bcast($\mathcal{T}_i$, *data*...) **bcast** is a version of **schedule** which, instead of placing the token message in the local scheduler, broadcasts it to each of the radio neighbors of the node running the current handler. The channel may well drop the message, and the DTM model does not assume ACKs in the communication protocol. This is a simple, low-level, single-hop broadcast, nothing more. More complex communication prmiitives are built up from here (see Section 3.5.1); even addressing messages to specific neighbors should be compiler controlled.

- is_scheduled($\mathcal{T}_i$)

- deschedule($\mathcal{T}_i$)

  is_scheduled and **deschedule** allow query and removal of token messages that are waiting in the scheduler. **is_scheduled** only reports the presence or absence of a given token in the scheduler—not the number or timing of token messages affiliated with that token. Similarly, **deschedule** removes all messages with a given token name.

- present($\mathcal{T}_i$)

- evict($\mathcal{T}_i$)

  The DTM does not specify how handlers interact with their own private memories—it is expected that they will have appropriate load and store operations—but it does specify the interface into the nodes' token store as a whole. **present** queries the local node for the presence of a token of a given name. **evict** removes the token, if present. When a token name T_i is evicted, the corresponding token object frees its private memory of length — no record of the token's presence is kept. If token handler tries to evict itself while its executing, the eviction happens as soon as the handler completes.

3.4 Token Machine Language

In this section we describe a concrete realization of the DTM model. We call this the Token Machine Language (TML). The DTM model provides the execution model; TML fills in a concrete syntax for describing handlers. As a whole, the systems' simplicity comes from the use of atomic actions as a concurrency model, and the unification of control, communication, and storage. And it is TML's simplicity that makes it valuable as a compilation target — a program is a collection of token handlers with a very disciplined interaction.

The core language used in TML for the bodies of handlers is a subset of C. This subset disallows pointers, procedure calls, and loops. Conditionals are still allowed; they make the execution time of a handler undecidable but statically bounded.

Of course, named fields such as `(int16_t x, int32_t y)` replace the undifferentiated blocks of bits described in the DTM model — both for the handler arguments (the token message payload) as well as for the token object memories and the shared memory. TML allows trailing arguments to handlers to be omitted with the understanding that they will take on a zero value. This allows some scheduling and broadcasts of messages to be more efficient. Data in the token object's private memory can be statically allocated by declarations of the form: "stored int x, y;" (initialized to zere), and data shared by all tokens as "shared int z;". Special syntax is added for the communication commands described in Section 3.3.2. Here is a sample of nonsense code showing what a token declaration looks like.

```
shared int s;

token Red (int a, int b) {
    stored int x;
    if ( present(Green) )
      x = 39;
    else {
      x += a + b;
      timed_schedule Red(500, s, a);
    }
  }
```

TML is currently implemented both as a simulator as well as a compiler targetting the NesC/TinyOS environment. In the TinyOS implementation, the DTM interface (send/schedule/etc) becomes a TinyOS module (DTM.send, DTM.schedule, and so on). The handlers become individual NesC commands. Our compiler for TML insures that only a safe subset of the NesC language is used—safe with respect to our DTM semantics. Yet there is nothing to stop the user from bypassing our TML compiler and manually writing code against our DTM library. They would be responsible for respecting the constraints of the DTM model, or at least breaking them in controlled ways.

Token Namespace in TML

The DTM model requires that there be a finite, but not statically known number of token in existence. The token store is *the* place for dynamic allocation, and any program that wants to make use of this needs a way to create an arbitrary number of tokens names rather than just those that occur in the program text.

Thus, we allow unique "subnames". If a program consists of token declarations for tokens {Red, Green, Blue}, the user may also reference {Red[11], Green[32]}. All subtokens of Red use the same token handler, but have their own token object and keep their own private memories. This does introduce a bit of ambiguity within each token handler. Which subtoken are we currently executing? For this reason we allow a syntax similar to component paramerization in NesC: "`token Blue[id]() { ...`". The variable "id" is in scope within the body of the handler, and refers to the numeric index of the "subname" currently invoked. Invoking without a subtoken index is the same as using 0.

Subtokens are somewhat like constructing multiple *instances* of a token "class". Again, a nodes' token store is its heap, and subtokens are the **only** form of dynamic memory available in TML. (There is not even the call stack normally available even in NesC.) Subtokens are thus a form of pointer in TML, but one that uses a consistent virtual address space across different nodes in the network. They can be used to allocate variable amounts of storage, or as will be seen in Section 3.5.1, to keep gradients from overlapping.

3.4.1 Returning subroutines

As our first example of systematically building up features with TML, we wish to add subroutine calls with return values. These are a staple of normal procedural programming, and are a necessity for us because the basic DTM model includes neither blocking calls nor built-in split-phased operations. We don't wish to have blocking calls in the model because our handlers must execute quickly for our atomic action model of concurrency to remain solvent, and recursive calls could take unbounded time to return. Besides, our model achieves its parsimony by eliminating the stack and having only one dynamically allocated structure (the token store). Moreover, we wish our token handlers to transparently support both local invocation and remote invocation; it would be out of the question to block on remote invocations.

But as a user, would *like* a token handler to be able to invoke another handler and use its return value, rather than merely be able to schedule its execution. We accomplish this by building subroutine calls on top of core TML by using a continuation passing style (CPS) transformation. This is our answer to the issue of split-phase vs. blocking operations. We circumvent the issue by having *implicitly* split-phase calls. The user simply uses "subcall" as below, and the CPS transformataton splits their handler into multiple handlers. The programmer must understand that when they're using this facility they may be breaking the atomicity of their handler — they are really using syntatic sugar for multiple handlers. This may require freezing all the live variables at the subcall-point and restoring that context again at a later time; thus there are possible efficiency concerns as well. As an example, consider the following simple code snippet.

```
token int Red(int a) {
  int y = 0;
  schedule Blue(4);
  y = subcall Green(3);
  bcast Red(a);
  return y;
}
```

The token handler for Red will be transformed so that it stops at the subcall to Green. It will allocate a continuation object in the token store. A new token declaration is added for this continuation. When Green is called it is passed the (sub)name of its continuation — which is the equivalent of a pointer to that continuation. The CPS transformation thus requires that every handler called via "subcall" take an extra continuation argument.

Performing a CPS transformation on TML is straightforward because of its simplicity and clear semantics, doing the same for general NesC code would be quite difficult. First, NesC has multiple execution contexts: tasks and events. Second, NesC has no easy mechanism for dynamically allocating the continuation objects.

CPS is well studied in the literature [18, 21, 41]. There are a number of optimizations that can make it efficient, especially in the case of TML where we may do whole program analysis. First, there are established techniques for minimizing the number of continuations created and the circumstances in which they must allocate memory. Second, in the case of TML we can optimize this subcall abstraction layer without breaking the core DTM semantics by allowing the implementation to make "direct calls" (i.e. NesC's "call") to non-recursive subroutines. This is similar to procedure inling; the compiler must choose when to make direct calls to a subroutine and when to go through the scheduler. In our case, since the execution time of each token handler is bounded by a estatically known quantity, we can compute the time cost of inlining a non-recursive subroutine call. We make this choice based on the constraints of the scheduling algorithm. For example, we may simply set a maximum desirable atomic action duration, and "inline" up to that maximum action duration.

Invoking split-phase TinyOS operations

Many operations in TinyOS are split-phase. In TML we can use our existing CPS transformation to give us a framework for turning TinyOS split-phase operations into TML "blocking" operations. Consider sensing in TinyOS which is split into a "getData" command and a "dataReady" event. In TML we use this to build a simple "sense" operation which gets treated just like a subcall. The handler is split, the first part becoming the "getData" portion, and the second part the "dataReady" portion. In this case, rather than the continuation being invoked from the user code, the first token handler runs to completion and the continuation handler is invoked by an *event* from the TinyOS subsystem.

However, we cannot give these event-invoked continuations the privelege of executing at any time whatsover, because we don't want to break the guarantees we've given through **timed_schedule** — they must go through the scheduler. Thus the when the underlying TinyOS event handler fires, it must schedule the associated token handler to execute. This token message can be given high priority, so that it executes quickly after the current atomic action completes, but it cannot be executed immediately. However, even this small amount of computation in the event handler takes some time. The TML scheduler will never be able to plan with perfect precision, it must assume some time-loss to events in the TinyOS subsystem; hopefully a predictable loss.

3.5 Example Applications of TML

TML is a compilation target. In this section we will add a gradient interface to TML. We do this by adding a very simple program transformation converting this "gradient language" to core TML, much like our implementation of the "subcall"

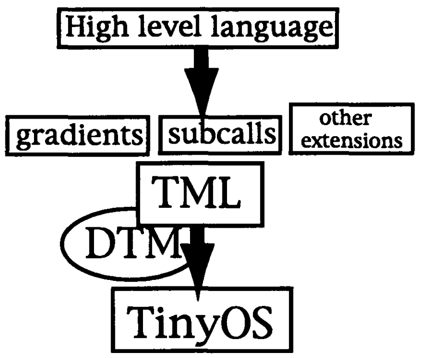


Figure 3-2: The ingredients in the TML system. DTM is the underlying abstract model implemented by TML.

keyword. What results is an collection of gradient network coordination operations that mesh naturally with the token-oriented semantics of TML. This is intended to represent typical usage for TML. After building up this gradient layer, we demonstrate its ease of use by writing a few simple applications. Finally, we discuss briefly how TML is used as a target language for the Regiment compiler.

3.5.1 Gradients

Gradients are general purpose mechanism for breadth-first exploration from a source node. A gradient establishes some sort of spanning tree which tells all nodes within the gradient their hopcount from the source as well as how to route to the source. See Directed Diffusion [24] for an example of gradients' utility and an overview of the design tradeoffs in gradient implementation.

Adding gradients to TML constitutes an extension to the language — or the creation of a new language, depending on how you see it — but at the same time it doesn't require modifying the TML core, which is a testament to the versatility of that core. Like our subcall facility, adding gradients involves a simple transformation in our compiler: implicitely appending extra arguments to token handlers. This time the extra arguments carry gradient information such as whop-count. We use an interface consisting of four operations.

- gemit($\mathcal{T}_i$, *data* . . .)

  Gemit is a version of bcast which begins a gradient propogation from the current node. A gradient is like a normal message broadcast with extra information. Gradient equality is determined by token name. Subtokens are used to achieve overlapping, non-interfering gradients using the same code (token handler).

- grelay($\mathcal{T}_i$, *data* . . .)

  Grelay is a version of bcast which continues the propogation of a gradient from the current node. Grelay fails silently if the named gradient has not been received.

- greturn($\mathcal{T}_{call}$, $\mathcal{T}_{via}$, $\mathcal{T}_{aggr}$, *data* . . .)

  Greturn allows data to be propogated up gradients to their roots, with optional aggregation along the way. A greturn call sends data up the "via" gradient, and fires the "call" token handler on the data when it reaches the source. The "aggr" argument should name a token handler accepting two arguments which can aggregate return values on their way to the same source, or it should be NULL for no aggregation.

- dist($\mathcal{T}_i$)

- version($\mathcal{T}_i$)

  We don't expose parent pointers through the gradient interface, because we don't wish for algorithms to depend on the details like single vs. multiple parents and so on. But we do wish the user code to be aware of distance from the source of a particular gradient. Further, if a gradient is reemitted from a source node — as is often the case — the user code should be able to differentiate these different gradient generations. To this end we allow the user to query the version of a gradient its received. This is first and foremost useful for doing initialization the first time a gradient is received. Both **dist** and **version** return -1 if the named gradient has not been received at the local node.

The gemit/grelay/greturn interface does not commit to a particular spanning tree selection or maintenance algorithm. Thus applications built against this interface need not commit to a choice. However, the developer will want to choose a gradient implementation appropriate to the application. Hence, the compiler should expose a set of choices of gradient implementation that covers the design space outlined in [24].

3.5.2 Timed Data Gathering

This extremely basic example shows how to use the above gradient interface to sample the light sensor on every node. It also uses a couple of trivial new keywords not mentioned above. The "startup" declaration indicates that the Gather and GlobalTree tokens will be scheduled when the node is first turned on. The "base_startup" keyword is similar, but only applies to the base-station node in the network. Also "BaseReceive" is pre-defined token handler supported only on the base-station, and used to return results to the outside world.

```
startup Gather, GlobalTree;
base_startup SparkGlobal;

token SparkGlobal() {
  gemit GlobalTree();
  timed_schedule SparkGlobal(10000);
}

token GlobalTree() {
  grelay GlobalTree();
}

token Gather() {
  greturn(BaseReceive,
            GlobalTree,
            NULL,
            subcall sense_light());
    timed_schedule Gather(1000);
  }
```

This program emits a gradient from the base-station, which relays itself until it reaches the edge of the network. Once a second, every node fires the "Gather" token which uses the globally present gradient to route data back to the base-station.

3.5.3 Distributed Event Detection

Consider the problem of local event detection with unreliable sensors. We can't trust the reading of a single sensor, but if some number of sensors within an area all detect an event, an alarm should be raised. Here we solve the problem by spreading out a small gradient from every node when it detects an event. When these gradients overlap sufficiently, the alarm is raised.

```
shared int total_activation;

token EventDetected () {
  emit AddActivation[MYID](1);
  schedule AddActivation[MYID](1);
}

token AddActivaton[sub] (int x) {
  if ( dist(self) < 2 )
    relay AddActivaton(x);
  total_activation += x;
  if (total_activation > threshold)
    greturn(BaseReceive, GlobalTree,
            NULL, ALARM);
  timed_call SubActivation[sub](1500, x);
}
```

```
    token SubActivation[sub] (int x) {
      total_activation -= x;
      if (total_activation <= 0) {
        evict AddActivation[sub];
        evict SubActivation[sub];
      }
    }
```

By using subtokens for AddActivation and SubActivation (indexed by the ID of the node emitting the gradient), we keep the different gradients from colliding. However, their overlap is still seen through the shared variable "total_activation". This demonstrates the utility of lightweight gradients spawned and destroyed from arbitrary points in the network.

3.5.4 Leader Election

As another example, we will now build a reusable leader-election component in TML. All the nodes that invoke ElectLeader($\mathcal{T}_i$) will participate in the leader election for token $\mathcal{T}_i$. One such node will eventually be decided leader and receive an $\mathcal{T}_i$ token. Multiple leader elections can go on concurrently in the network; this is because ElectLeader($\mathcal{T}_i$) uses subtokens indexed by $\mathcal{T}_i$ for all of its computation. The problem of garbage collecting dead tokens is ignored for this example.

```
shared int winner;

token elect_leader(tokname T) {
  int current = winner;
  if (current == 0 || current < MYID) {
    winner = MYID;
    timed_schedule Confirm_Fire[T](5000, T);
    emit Compete(MYID, T);
  }
}
```

```
token Compete(int id, tokname T) {
  if (winner == 0) {
    winner = MYID;
    timed_schedule Confirm_Fire[T](5000, T);
  }
  if (version(Compete) == 0 || id > winner) {
    winner = id;
    relay Compete(id, T);
  }
}

token Confirm_Fire[sub](tokname T) {
  if (MYID == winner) schedule T();
}
```

3.6 TML Discussion

TML is currently implemented as a high level simulator and as a compiler targeting the NesC/TinyOS environment. The mapping from Token Machines onto NesC was discussed in Section 3.4. Overall, it took relatively little effort to map TML into TinyOS because TML is not a *mechanism* so much as a *discipline*.

Our current TML implementation has some shortcomings with respect to the features laid out in this chapter. Namely, the current implementation implements subcalls but makes all subcalls "direct" (as described in 3.4.1), and thus circumvents the necessity of the CPS transformation. As a result, we must manually insure that handlers complete in a relatively short time. This part of the implementation will be corrected in the near future.

Code size for compiled TML code is very good. Only a small constant factor size increase is added to the TML source when translated to NesC. The run-time support (DTM component) is also relatively lean. When compiled for the Mica2 mote, it consumes only 8836 bytes of ROM. RAM usage is worse: 817 bytes, including a token store of 320 bytes. Both RAM and CPU usage suffer as compared to "equivalent" native TinyOS code. This is because of the overhead of running the scheduler component and unnecessary copying of buffers. We believe that there are many optimizations yet to be exploited which can reduce memory redundancy. Future work will be move in this direction.

What we have learned thus far from our use of TML is that its two important qualities are: the atomic action model of concurrency, and the fact that communication is bound to persistent storage (tokens). The former precludes deadlocks and makes reasoning about timing extremely simple. The later essentially gives us a way to refer to communications that have happened through the token they leave behind. Also, tokens give us some of the benefits of "viral agent" type models of ad-hoc distributed computing (such as in [13]), without the overhead. They can be seen as a lightweight version of this agent-oriented model.

Chapter 4

A Regiment Compiler

The current prototype compiler for the Regiment language supports a restricted subset of the language specified in the second chapter. In short, the currently operational portion of Regiment includes the region operations, but lacks proper type-checking, full stream rate control, and operations over basic types other than 16 bit integers.

The compiler is written in Scheme. It converts Regiment programs into Token Machines (TML programs as described in the previous chapter). The programs shown in this chapter appear exactly as processed by the compiler — a Scheme syntax is used — but the operations are the same as chapter one. The output Token Machines are subsequently compiled into complete NesC programs by a back-end written in Haskell. The next implementation of the Regiment front end will be entirely written in Haskell and will leverage the Glasgow Haskell Compiler's type checker.

Because the compiler transforms a Regiment program into a complete NesC program, it differs from most query languages. In a distributed database, queries are disseminated to all nodes for processing by local query processors. These query processors are akin to program interpreters. Regiment "queries" are fully compiled rather than interpreted at the nodes. This decision was made at the outset, based on the observation that performing all the computation necessary to interpret Regiment queries in-network would put too large a burden on the sensor nodes (in terms code size as well as computation). Instead, we want all heavy-weight processing of the Regiment macroprogram to happen in the compiler, and the tiny sensor node to receive only a lightweight Token Machine to execute. Future Regiment back-ends may choose to reverse this decision. Advantages in doing so might include more compact representations (queries vs. binary images) for dissemination, and easier dynamic reprogramming.

4.1 Regiment Compilation: Informal Intuition

Before diving into the innards of the Regiment compiler this section gives an intuition for the priorities and the process of Regiment compilation.

The precise operational semantics of programs from Section 2.3.5 should not be immediately clear — Regiment's strength is that its semantics are high level. Hence there are many operational realizations that would match the denotational semantics. Our choices are thus driven by general principles of robustness and efficiency:

1. **Communicate locally:** minimize situations where the base-station(s) must be brought into the loop.
2. **Survive point failures:** there should be no "critical nodes" in the network.
3. **Avoid hot-spots:** load should be distributed evenly across nodes.

Now, as an example, let us consider the compilation and run-time behavior of the tracking programs of chapter two. The code for these examples appears on pages 27 and 27. Considering first the non-sentried version: the compiler, analyzing the control flow of the Regiment program, knows that every node will need to monitor its proximity sensor to determine its membership in `selected`. Thus, once the Regiment program has been loaded on the sensor nodes, no communication is necessary to form the region `selected`. Next the `cluster` operation is applied to form the region `globs`. Cluster need only perform local communication in the network. Essentially, a gradient is spread among members of `selected`, expanding from every node at once but stopping when non-`selected` nodes are encountered — thus forming connected components. The gradient is started from every node, but merges upon collisions, resulting in a single gradient and a single leader elected (the cluster head).

Thus, without anything but local computation, we have a network full of small contiguous regions ("globs") around each of the tracking targets (assuming no two targets are close enough to interfere). The next stage of our computation involves computing `amap centroid globs`. In this example the control flow of the high-level macroprogram is known; therefore, each glob knows that the next thing that will happen to it is processing by `centroid`. Thus without any outside instruction (from the base-station) each glob will begin executing `centroid`.

Examining the definition of `centroid` we see that it consists a fold over the input region, followed by mapping a function over the resulting stream. It so happens that each glob from `cluster` was formed by a spreading gradient. The compiler knows to use that gradient to do the aggregation requested by fold.

Once the fold is accomplished the globs aggregated into streams. These streams have ended up at the leader nodes of each glob's gradient. This leader node applies the mapped function (seen in the definition of `divide`) to each element of the stream. Again, here is a place for future improvement. The current implementation greedily computes the mapped function wherever the stream first becomes available (at the cluster head). However, if the mapped function happens to be compute intensive, it would be better to spatially spread out its computation; perhaps along the route to the streams' next destination.

Where is the streams next destination? If we are one of these cluster heads, how do we know where to route it? Again, the compiler has already told us. Because `amap centroid globs` was the final return value of the Regiment program, that means the user desires the value of that expression. To extract it, the data stream must be routed to the base-station. The current Regiment implementation maintains a single global gradient to enable any node to communicate with the base-station. (Likewise, having multiple base-stations would be as easy as maintaining multiple gradients for them.) The cluster heads use this global gradient to route the stream of approximate target locations back to the user. Of course, everything we have described so far is being dynamically recomputed. The targets move, regions shift, and cluster heads move with them. This is a major source of robustness. For example, a cluster head failure only lasts until the next gradient refresh.

Finally let us consider the sentry enhancement to our example program. We are running the almost the code as before, but setting up an event to serve as a barrier on the computation of `targets`. How does the control flow for this new program work? The final return value of the program is `until event nullArea handler`. No work is required to compute `nullarea` (the region equivalent of the empty set). The compiler knows it needs to compute the event first. Doing that requires forming the border around the world. This invokes a black-box process that unfortunately touches the whole network (but at a low frequency). The nodes selected to be part of the border are notified, and accordingly they continuously monitor their proximity readings to determine whether or not to "fire".

When any of them detects the event locally, it knows the global event has fired. (That's what `whenAny` means.) Where should that node route the information? In the current implementation all event firings go through the base-station, but this is a place for improvement. An ideal implementation would route an event it exactly to those portions of the network that depend on it. (In this case, if we had a model of the movement of the objects being tracked, perhaps we might notify only the portion of the network in close proximity to the border, rather than waking up the whole network.)

When the base-station learns of the event, it puts must shut off the border region and activate the normal tracking program. That is, stop computing `nullArea` and invoke `handler` on the resulting event which will begin computing `targets`. This will require activating the process computing `targets` across the entire network. Subsequently, the rest of the computation looks exactly like the non-sentried version.

4.2 Current Language and its Restrictions

Now we will examine a the subset of the Regiment language currently implemented. Then, in the following sections, we will go over the compiler pass by pass.

More so than most languages, Regiment faces critical choices as to which portions of the computation should happen statically ("compile time") and which dynamically ("run time"). With Regiment's semantics, these choices don't generally affect the meaning of programs (just their implementation), but may change the space of valid programs.

Regiment faces a strong impetus to make features static rather than dynamic; efficiency is critical in sensor networks. Dynamic features directly translate into energy cost for the sensor network. Thus, for the time being we have pushed as much as possible into the static domain (offline). For example, rather than allowing higher order functions and the associated highly unpredictable control flow at *run*-time, Regiment's current implementation is multi-stage. It includes a static elaboration phas. In it, all procedure calls to user defined functions (with monadic return values — Stream, Area) are inlined. This unrolls all the loops in the program having to do with distributed operations. The transformation will be described in detail in Section 4.4.

Static elaboration does not change the semantics (types, evaluation rules) of the Regiment language, it merely opportunistically pushes evaluation forward into compile time. Yet the elaboration phase has real value because it allows users to gain much of the benefit of procedural abstraction and bounded recursion without having to execute those behaviors on the distributed run-time of a resource constrained sensor network.

There is a well-established literature on strongly typed, multi-stage programming languages [44, 42]. Also related is general work on partial evaluation [25], macro systems for programming languages [19], and static-elaboration for hardware design languages [7, 33]. In particular, the current Regiment implementation makes the same design choice as Bluespec — it uses software expressivity for building the *structure* of Regiment queries. However, because sensor networks are more dynamic than hardware designs, we may allow more features to enter the dynamic realm as we develop better techniques for executing traditional software constructs effectively within the network.

For the rest of the thesis we will refer to the following distinct phases in the compilation and execution process:

- Static elaboration (compile time)
- Query dissemination (load time)
- Query execution (run time)

A further restriction on the current version of the Regiment Compiler is that it uses a simple monomorphic type system. 16 bit integers are the only simple type. This is merely a limitation of the current implementation. Eventually Regiment will have a full Hindley-Milner type inferencer and a type system resembling Haskell's [26]. Also, the language specification described in chapter two includes frequency-control operations, but those have been only partially implemented in the current compiler. Fidelity feedback (the percentage of processors in the region responding in a given time-step) is not implemented at all, and will require significant additional effort.

4.2.1 Query Circuits

Because Regiment uses a static elaboration phase to eliminate most control structures in the language, the resulting program is best visualized as a *query circuit* — a network of stream/region processing operators. For the rest of the chapter when we refer to the "network" we will be talking about this dataflow graph of operators, rather than the run-time network of sensor nodes. The operators in the network are Regiment primitives such as `smap`, `amap`, `afold`, `afilter`, and `until`. The job of the Regiment back-end is to compile such a query circuit into an efficient node-level program. A simple query circuit is shown in Figure 4-1. For brevity, the nodes f and g appear in lieu of actual function definitions.

We call all non-monadic valued operators (drawn with rounded edges) *local* and all others *distributed*. The idea is that local operators, when evaluated, occur only within the scope of a single processing node, whereas distributed values require communication and coordination. Regiment query circuits differ from those seen in traditional

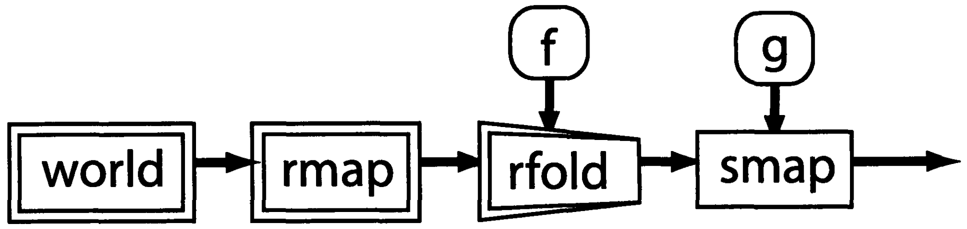
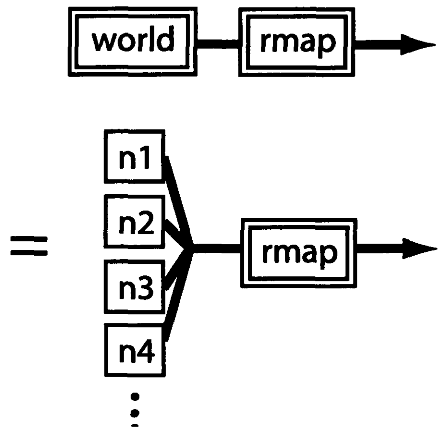


Figure 4-1: A simple query circuit. Double-boxed operators take and return regions, single boxed ones streams, and afold reduces regions to streams. Operators returning events are outlined with hexalateral boxes, and operators returning basic (non-monadic) values have have rounded edges.

stream processing systems [1, 5, 51] because of the presence of region-producing operator nodes (double boxed). These nodes can be thought of as representing an unknown quantity of (single boxed) signal-producing sources — a region is a collection of streams. As illustration, consider the figure below; it equates an `rmap` over `world` with an `rmap` over an unknown number of individual node data streams.



4.2.2 Evaluation Ordering and Query Circuits

In the current implementation, the data in the streams/regions flowing through a query circuit are always *pushed* rather than pulled. The question, however, of when streams are *initiated* is a separate matter. It has to do with the *evaluation order* of the high-level Regiment language. Regiment's semantics specify a lazy-evaluation strategy. However, as with any call-by-need language, whenever it is certain that an expression *will* be evaluated it is immaterial *when* it is evaluated. For Regiment programs expressing relatively simple queries all or most of the programs *distributed* operators generally falls into this category and can be evaluated at the outset. Only operators inside lambda expressions have their evaluation delayed, and because all monadic-valued user defined functions have been inlined, this leaves only distributed operators inside the definitions of (monadic-valued) function arguments to higher order primitives (`amap`, `afold`, etc). These are relatively rare in Regiment, but do occur — primarily as a result of event handling and clustering.

Event handling delays evaluation of distributed operators. The `until` construct takes an event, an initial stream/region, and an event handler. The event handler must be a function that consumes the value produced by the event argument and returns a new stream/region. Thus distributed operators in the handler function (which must exist) are data-dependent on the value produced by the event. This constrains the evaluation order of `until`'s subexpressions. The Regiment runtime must acquire the event's value at some location, and route this information to wherever it is needed for the formation of the subsequent stream/region. However, if the event handler (or some part of its result) does not depend on the particular value carried by the event, we might speculatively evaluate that stream returned by the handler, and simply discard its results up until the point of occurrence of the event.

Clusters are another common cause of functions that return region values. The reason is that they generate region-of-regions. For `(amap f (cluster r))`, the function `f` must (to do anything non-trivial) have a monadic return value and therefore contain distributed operators. However, this usually does not cause painful control flow or communication — generally working with clusterings only requires communicating within the local connected components.

4.3 Compiler Structure and Simulator

Our compiler follows a micro-pass architecture. That is, it is composed of many small compiler passes. The output pass of each is a *runnable* (simulatable) intermediate form. The passes are grouped into three broad phases. The first phase of the compiler consists of preprocessing and normalization, the second involves analysis and annotation, and the third performs code generation and optimizes the result.

We use the Scheme macro-system [27, 16] to build a series of embedded languages, making output of each pass runnable code. This is a multi-resolution simulation system. The earlier passes produce code that when executed run an extremely crude simulation of the Regiment macroprogram. Simple list data-types represent regions and areas and there is no decentralization of control-flow. This first-level simulation is abstract and inaccurate, but is still surprisingly helpful for quickly getting a sense of what a Regiment program means.

The second broad phase of the compiler works with the now-simplified Regiment program (query circuit) to analyze and annotate it. These passes also use the same form of crude, centralized simulation. (They don't change the structure of the program, merely annotate it.)

The most substantial portion of the compilation process occurs at the outset of the third phase of compilation. It is there (in "deglobalize") that the program is converted from a macroprogram to a Token Machine — a node level program described in chapter three. When run, these Token Machines invoke a basic network simulator, called Simulator0 (pronounced "Simulator-Nought" — the first simulator for Regiment). This is also implemented in Scheme. It is multithreaded, running a thread for each node in simulated network. However, it uses an extremely simplified communication model (disc radio model), and does not model node failures. For more realistic simulation, the user runs the remainder of the compiler, produces NesC code, and simulates it with the TinyOS simulator (TOSSIM), which has a bit-level-realistic radio model.

4.4 Static Elaboration

All programming languages face tradeoffs in terms of choosing which features will be static and which dynamic. In varying degrees this applies to typing, memory allocation, procedure calls, method dispatch, and so on. As mentioned above, for the current Regiment implementation we choose to restrict all (monadic-valued) user procedure applications and memory allocation to the the static elaboration phase. The Regiment static-elaborator proceeds by inlining all function applications where the code for the operator is available. *However*, there is an exception that expressions which do not and cannot have the Stream monad in their type *do not* need to have all their user functions inlined. (Note that the Region monad includes the Stream monad.) The intuition is that the user may run whatever computations they want as long as they stay local to a single sensor node; and as long as there is no Stream monad in the functions type signature, there can be nothing but local computation and storage. For example, the user may wish to calculate the recursive factorial function. This function can then be mapped over a stream, but it itself returns a non-streamed result and thus it is no concern to us in doing our static elaboration.

Further, the compiler demands that when static elaboration is complete that all function arguments to higher order primitives are *known code*, that is, the compiler knows which function definition (lambda expression) the expression refers to. For the purpose of the current compiler, this means that the expression is either a lambda expression, or a let-bound variable reference to a lambda expression. Lambda-bound variable references are conservatively classified as *unknown*. Thus, by fiat, the output of the static elaboration phase is required to satisfy the following constraints.

1. The only procedure applications are primitive applications (rfold, rmap, etc) (with the possible exception of non-stream typed functions).

2. All procedure-typed arguments to higher-order primitives are to "known" closures.

3. No memory allocating primitives (cons) remain.

```
 (letrec ([f (lambda ...)])
      (letrec ([loop
               (lambda (n)
                 (if (= '0 n)
                     world
                     (rfilter f (loop (- n '1)))))])
        (loop '4)))

Statically elaborates to:

  (letrec ([f_1 (lambda ...)])
     (rfilter f_1 (rfilter f_1
        (rfilter f_1 (rfilter f_1 world)))))
```

Figure 4-2: A simple program before and after static elaboration.

This restricts Regiment to a subset of the language of chapter one. The memory allocation limitation is merely a limititation of our current implementation.

As an example, refer to Figure 4-2. The input program uses recursion in a monadically-valued function (`loop`). The static elaboration unrolls this loop, producing code that has no non-primitive applications, and where the only function-valued operand (`f_1`) is a "known function" (`(lambda ...)`).

4.5 Preprocessing and Normalization

The core work of the Regiment compiler is in transforming the macroprogram to a node level program. Before this can happen the compiler must do a routine simplification of the input program. This happens in a number of steps which are summarized briefly here. These transformations are standard fare and not specific to Regiment; thus their details are not pertinent to the thrust of this thesis.

- **verify-regiment**: verifies syntactic properties of the Regiment program and performs rudimentary type checking.[1]

[1]TODO: We don't have full polymorphic type-inference/checking in the current compiler.

- **eta-primitives**: makes sure that all references to primitive functions occur in operand rather than operator context. That is, primitives' names cannot be used as values. This pass accomplishes this by eta-expanding references to primitives' names into lambda expressions.
- **rename-var**: renames all variables in the program so that each variable has a unique name.
- **remove-unquoted-constant** ensures that all literals are preceded by a quote: "'3".
- **reduce-primitives** re-expresses certain Regiment primitive functions in terms of more basic ones. For example *circle-at* is just shorthand for a call to *circle* combined with a call to *anchor-at.*
- **uncover-free** identifies and labels free-variables at each lexical scope.
- **lift-letrec** flattens the program so that it contains a single top-level letrec.
- **lift-letrec-body** relocates the body of the top-level letrec expression into a new letrec-bound variable binding. Hence, the body of the top-level letrec expression becomes simply a variable reference.
- **remove-complex-opera*** This orders all operations by introducing variable bindings for all intermediate computations. Subsequently all operands to primitive functions are either literals or variable references.
- **verify-core** Verifies that the intermediate program conforms to appropriate specification. A grammar for this "core" Regiment is given in 4-3.

4.6 Analysis and Annotation of the Macroprogram

With desugaring complete, the next several passes analyze the high level program and attach various kinds of information to it. All of this information will then be used by the "deglobalize" code generator.

```
⟨program⟩ ⟶ ⟨letrec⟩
⟨letrec⟩ ⟶ (letrec ([⟨identifier⟩ ⟨exp⟩]*) ⟨exp⟩)
⟨exp⟩ ⟶ ⟨constant-exp⟩ | ⟨identifier⟩
    | (if ⟨exp⟩ ⟨exp⟩ ⟨exp⟩)
    | (⟨primitive⟩ (⟨exp⟩*))
    | (⟨exp⟩ (⟨exp⟩*))
    | (lambda (⟨identifier⟩+) ⟨letrec⟩)
```

Figure 4-3: Grammar for core (desugared) mini-language. The added restrictions are that applications of non-primitive operators have non-monadic return values, and that function-valued operands are refer to known function definitions.

4.6.1 Add Heartbeats

As alluded to in chapter three, during run-time regions become distributed, collective firings of token handlers. Different regions in the network may fire at different frequencies. For example, we may wish to sense light readings once a second and temperature readings once every two seconds. This pass analyzes the program and attaches default frequencies to different operators in the program. For now it uses a simple heuristic. Region *formation* in terms of geographical or network-topological entities should proceed on a slower time-scale than sensor reading and computation. It also employs user frequency annotations as constraints. For example, the following would be used to read the light and temperature sensors at different rates.

```
let lights = set_freq 1.0 (map sense_light world)
    temps  = set_freq 2.0 (map sense_temp  world)
in ...
```

Eventually, stream rate controls should be more dynamic, so as to adapt to changing circumstances (for example, power levels). Presently, they are statically scheduled.

Regiment's Timing Model

While we refer to "streams" throughout this thesis, Regiment has not semantically committed itself to discrete streams. For example, it has no sliding-window operator.

Ideally, we think of Regiment's streams as continuous signals which are only approximated by their discretized counterparts. In the future we will look into incorporating more signal-processing functionality as suggested by this viewpoint. For the time being we must bear this in mind when considering Regiment's timing model.

For example, Regiment has a thus-far undiscussed (and incompletely implemented) `smap2` operator which joins two streams together, as well as an `rmap2` which joins the data in two regions over their intersected area. How does this work when the streams or regions are of different frequencies? May we pair stale values of one stream with the fresh values of another? Can we smooth the data? Regiment currently doesn't answer these questions. It conservatively increases frequencies to match each other when made necessary by joins.

4.6.2 Add Control Flow

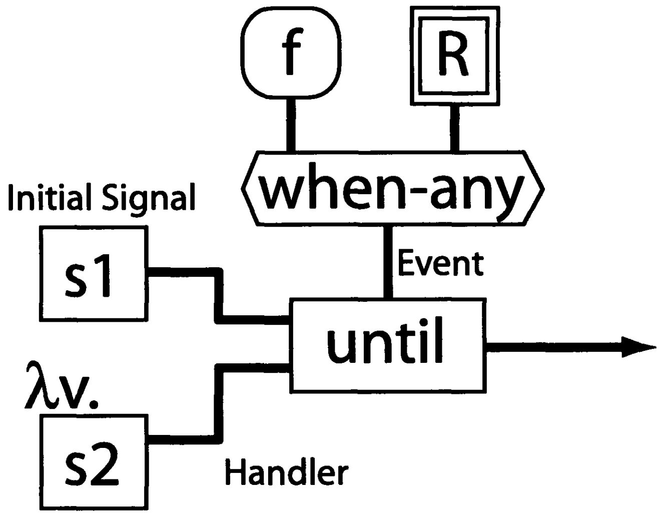
Regiment's purely functional semantics enable it to adopt a lazy evaluation model. However, purely functional languages are in some senses "immune" to differences in operand evaluation strategy [40]. Thus, like other purely functional languages, Regiment tries to be eager whenever it can. In practice almost all Regiment programs have an eager evaluation order, as we will see, it is only event handling and inadequacies of our dataflow graph (due to conditionals, and regions-of-regions) that cause us to resort to laziness.

In Regiment, order of evaluation becomes analogous to push vs. pull communication in the networking layer. If we have a simple network of rmaps, rfolds and the like (without conditions or events) control flow may start at the leaves of the program and come towards the root, and data can be streamed along with control in that same direction. If we introduce an event and an `until` statement, then we face a situation where the macroprogram can only evaluate the event-handler when the events value is available.

Conditionals present a similar problem because their nondeterminism results in us having references to regions whose identity is not statically known. In such a scenario the region formation code must wait to be executed until the correct region is issued a dispatch at run-time (which also requires potentially costly communication).

The job of this pass, then, is to analyze the operator network and add control flow edges, also labeling region and stream carrying edges as *push* or *pull.* (Pull meaning simply that the appropriate control flow edges must be invoked to evaluate the parent expression).

As an example of a pull scenario, let's consider again the `until` construct.



The `until` construct takes an event, an initial stream/region, and an event handler. The event handler must be a function that takes the value produced by the event argument and returns a new stream/region. When the until expression receives control, it transfers it to its event and initial-stream arguments simultaneously. Upon the event firing, the `until` node transfers control to its event handler.

Consider the scenario in which we are compiling a top-level `until` expression. Control transfers immediately to the event and initial signal expressions, allowing them to have push edges for their distributed operators. The interior of the event handler may also have distributed operators with push edges. But the event handler itself (the `lambda` node) will be annotated with a pull edge, indicating that the `until` node must initiate execution of that `lambda` node — "call" the function, which in this case involves routing the events value to he appropriate place(s) in the network for the event handler to generate its result.

4.6.3 Add Virtual Places

The Regiment program is executed in a distributed environment. Therefore, values traveling through edges in the query circuit are located at different points in physical space. This is similar to typical distributed stream processing, except more difficult for a couple of major reasons. First, communication is not all-to-all; we have to work with mesh communication only. Second, regions exist not at just a single point in the network, but at an unpredictable collection of many points simultaneously.

Nonetheless, the Regiment compiler can infer some information about the *places* that different values traverse. For example a region resulting from a filtration or an intersection operation is a subset of the original region. This is extremely relevant for doing communication efficiently — i.e. you can use the parents spanning tree rather than flooding the sensor network. This pass proceeds by introducing a namespace of virtual places, then inferring relationships (equivalence, subset, member) between them. It gathers information by leveraging straightforward knowledge about the basic operators (i.e. `amap` doesn't change the location of the region).

4.6.4 Add Routing

With our virtual place annotations in place, we can look for edges whose sources and sinks don't exist at the same virtual place. When this occurs, routing is necessary. Hopefully, the previous pass has left us with the information we need to conclude "you can use spanning tree X to communicate to region R". Otherwise, we might have to resort to flooding the communication network to reach a region, or to doing a breadth first search to find the sink for a stream. This pass decides which of these communication mechanisms will be used, and attaches the information to the appropriate edges in the query circuit.

4.7 Deglobalize

To summarize, upon reaching this phase of the compilation process we have: expanded the program into a network of stream/region operators, annotated it with timing information, made push/pull decisions for all of the stream-carrying edges in network, adding a separate set of control flow edges, and, where necessary, labeled the sources and sinks of these edges with information on spatial redirects (routing).

The final major pass in Regiment's compiler is *deglobalize.* It's name takes after the fact that the reduced and annotated streaming dataflow graph is still a "global" program — it makes reference to non-local data structures (regions). The output of deglobalize is a complete Token Machine program, conforming to the TML language and execution model described in chapter two. Now we will conventions used for the output TML program, and the code-generation process that gets us there.

4.7.1 Formation and Membership Tokens

The first convention we establish is that every edge in the input graph which carries a value of type stream (including regions), will have an interface as follows: a *formation* token that initiates the creation of the stream and a *membership* token that constitutes an event signaling membership in the region or reception of a value in the stream. We will allow ourselves to refer to the formation and membership tokens of an operator node, which really refers to the corresponding tokens for its outgoing edge(s).

A formation token may be invoked once to create a cascade (stream) of membership events. Or it may be reinvoked at each step in time. This depends on the operator generating the particular stream in question. But in general, for regions formation is a process occurring on a time-scale slower than membership. That is, the selection of a subset of nodes happens at a relatively slow time scale, whereas reporting of samples from those nodes happens much more rapidly. Of course, for a given region, formation events and membership events needn't happen in the same place. A region is a network *process* that is seeded through formation tokens at some set of locations over some time window, and whose formation signals another set of membership tokens over another set of locations and a later time-window.

Events have membership tokens which fire only once (for that particular event value). Events are constructed by predicates that watch streams. At the token level this becomes simple. The input stream to a `whenAny` primitive (`whenPercent` is not presently implemented) causes a sequence of membership events, when each of these occurs, the predicate is checked; if it is satisfied the events membership event occurs. The difficulty with events comes in routing the information appropriately.

For anchors (and also streams) the membership token will only fire in the node elected as the anchor. As this node changes over time, the firing location for this membership token follows it. Presently, anchor nodes are stateless so the migration poses no problems. In the future, the run-time will attempt to migrate the state associated with the anchor's tokens.

4.7.2 Per Operation Code Generation

This section will describe the code generated for several of the major operators in Regiment. For brevity, only a subset of the full Regiment primitive library are described.

1. **amap** takes one area argument and one argument thats a known function. There are three possibilities as to how the operator is wired:

    - If the operator node producing the region value is known, and the associated incoming edge is a *push* edge, then the membership token of that edge is wired directly to the formation token created by the of the amap node. That is, code is added to the token handler for that membership token, causing it to invoke amap's formation token. The formation token invokes the mapped function, and the membership token fires upon its completion and carries the return value of the function.

    - If the operator node is known, but the incoming edge has a *pull* orientation, then the amap primitive is responsible for invoking evaluation that corresponds to Regiment expression signifying region operand. We refer to the control flow graph, which includes information as to which leaf nodes need to be seeded control (which formation tokens need to be invoked, and *where* they need to be invoked). The formation token of the amap node is wired to the formation tokens of these leaves.

    The membership token for the amap's incoming region-carrying edge is therefore not wired to the amap's formation token, but instead directly invokes its function argument. As before, upon the completion of this function the amap's membership token fires and carries its value onward to the next stage of the computation.

  - The operation producing the input region is not known. Even with Regiment's static elaboration phase this may occasionally happen. Regions can still be passed as arguments to functions. Thus we may be faced with a variable reference to an unknown region. In this case, at run-time we will receive the *name* (a reference) to the region that the amap applies to. We must then broadcast amap's formation token to all nodes holding the membership token associated with that named region. This is a spatial redirect. Ideally, we can use "place" metadata added by a previous pass to optimize this multicast operation. However, the current implementation handles this scenario simply by flooding the message throughout the network, hitting all members of the named region in the process.

2. **smap** works in the same manner as amap. Currently smap also processes data "in place", i.e. where it becomes available in terms of its membership token firing. Smap operators should have more flexibility than amap operators, not being bound to a region. (Streams are more mobile than regions.) In future implementations we will look into intelligent placement of smap operators inside the network. For example, it might be placed at the sink(s) of the stream rather than the source (or anywhere in between).

3. **afilter** is also similar to amap. It is essentially the same as amapping the identity function, with the caveat that the outgoing token event of the afilter

node is only fired if that particular element satisfies the input predicate function. Thus the membership events coming out of an afilter will happen at a subset of the locations of those coming in.

4. **afold** Starting from one or more nodes hosting formation token events, afold spreads a gradient that covers the region in question (thereby establishing a spanning tree for that region). Each subsequent formation firing refreshes the gradient (adapting to changes in communication topology). When the spanning tree is established, members begin reporting data up the spanning tree. This happens at a frequency set by the add-heartbeats pass, and the reporting loop runs independently of and asynchronously to the formation/gradient-refresh loop.

   Aggregation happens automatically as data travels up the spanning tree. The outgoing edge for the afold operator is a stream, and when the final aggregated datum arrives at the root of the tree, the membership token for this stream is fired at that root node.

   The current afold implementation could be substantially more robust. There are well established techniques for robust aggregation that we will employ in future refinements, for example, keeping multiple parents or aggregating the final result to more than one node.

4.7.3 Returning Data to the User

One other relevant point concerns the final outgoing edge in the query circuit — the edge that carries data to the user. The operator that is in this final position is treated much like it would be elsewhere, but with the exception that the result is streamed through a global spanning tree to the base-station (of which there are one at this point). If a region rather than a stream is returned from the network, this necessitates that all the nodes within than region report their data back through the global spanning tree, which is a costly operation, so usually we try to do most of the reduction on regions inside the network.

4.7.4 Simple Code Generation Example

Now we illustrate the above techniques with a simple compilation example.

```
(afold max 0 (amap sense_light (rhood 5 (anchor-at '(30 40)))))
```

This example simply chooses an anchor node in the geographical vicinity of coordinates (30, 40), and builds a region consisting of all the nodes within 5 network hops of that anchor. (`Rhood` is a shorthand for `radio_neighborhood.` . When that region is formed, the light readings are extracted from all the sensors inside the region, and finally the maximum light reading is returned.

For this program, control flow starts in anchor-at and passes outward through `rhood`, `rmap`, and finally `rfold`. Deglobalize adds extra tokens to establish the global routing tree and to stimulate the leafs of the program (anchor-at). The output of the deglobalize pass is shown here.

```
(deglobalize-lang
  '(program
     (socpgm (call spread-global))
     (nodepgm
       (startup leaf-pulsar_tmpanchorat_7)
       (tokens
         (spread-global ()
           (emit global-tree)
           (timed-call 1500 spread-global))
         (global-tree () (relay)))
         (leaf-pulsar_tmpanchorat_7 ()
           (call f_token_tmpanchorat_7)
           (timed-call 1000 leaf-pulsar_tmpanchorat_7))

         (f_token_tmpanchorat_7 () (flood constok_18))
         (constok_18 ()
           (if (< (locdiff (this-loc) '(30 40)) '15)
          (elect-leader m_token_tmpanchorat_7 getdist_19)))
      (getdist_19 () (locdiff (this-loc) '(30 40)))
      (m_token_tmpanchorat_7 () (call f_token_tmprhood_8))

      (f_token_tmprhood_8 () (emit m_token_tmprhood_8))
      (m_token_tmprhood_8 () (if (< (dist) '5) (relay)))
      (m_token_tmprhood_8 () (activate f_token_tmprmap_10))

      (f_token_tmprmap_10 ()
        (call m_token_tmprmap_10 (local-sense 'light))
        (timed-call 100 f_token_tmprmap_10))
      (m_token_tmprmap_10 (v) (call f_token_result_6 v))

      (f_token_result_6 (v)
        (greturn
          v
          (to m_token_result_6)
          (via m_token_tmprhood_8)
          (seed '0)
          (aggr tmpfunc_11)))
      (tmpfunc_11 (a_3 b_2)
        (let* ([result_5 (max a_3 b_2)]) result_5))
      (m_token_result_6 (v) (soc-return v))
)))
```

The socpgm clause contains code that is only executed by the base-station (Source Of Control). In this case the base-station invokes a local copy of the spread-global token, which takes no arguments, launches a gradient, and loops after a time. Please refer to chapter three for definitions of the gradient operations (emit, greturn, etc). The global-tree gradient simply relays itself until it hits the boundary of the network. It doesn't need to invoke any other tokens, its their simply to provide a spanning tree used by greturn.

You can also see a startup clause which launches a token in every node of the network at startup time. The leaf-pulsar tokens are there to periodically send pulses to nodes at leaves of the control flow graph. This calls the formation token for the anchor node (f_token_tmpanchorat_7 — formation tokens begin with f_ and membership tokens with m_).

The anchor-at operation uses a relatively inefficient technique for electing an anchor. Rather than using a geographical routing protocol (which would be expected in a refined deployment with localization capabilities), it floods the network with a "consideration token" and all nodes within a given fixed distance of the target location are considered for a leader election. Flood is a macro that subsequently expands into an extra token handler emitting a gradient. Elect-leader is another macro that expands into a bundle of token handlers in subsequent passes. The extra argument getdist_19 provides a function to minimize. Of the nodes participating in the election, the one with the smallest distance from the geographical location will receive the token m_token_tmpanchorat_7 — the membership token for the anchor.

This membership token immediately invokes the formation token for the k-neighborhood, without a spatial redirect. (Later Token Machine optimization passes eliminate this extra call by inlining the call and eliminating dead code.) The anchor becomes the epicenter of the gradient that forms the neighborhood. This gradient expands to a certain number of hops and then stops. All the nodes within that radius have then received rhood membership tokens.

Now the region has been formed. Next we'll begin taking light readings. But we want that to be its own loop running at its own frequency. Hence the activate call; it is a macro that checks if a token is already scheduled, if so does nothing, otherwise schedules it immediately. This can be used for prodding a recursively looping token to make sure it's alive, but without interfering with its timing cycle.

The f_token_tmprmap_10 does the sensing and passes the result on to its membership token, which in turn activates the next step in the chain: the afold's formation token, which is called f_token_result_6. For this aggregation, the compiler knows that there's already a skeleton underpinning the region (because of the gradient emission forming the k-neighborhood). Thus the greturn uses the spanning tree from `m_token_tmprhood_8`. The aggregation uses `tmpfunc_11` to combine results along the tree, and when it gets to the root it invokes the `m_token_result_6` token on the final aggregated value. All that's left for this token handler to do is stream the answers back to the base-station. `soc-return` is another shorthand for a greturn statement that uses the global tree and returns the results to the user.

Chapter 5

Conclusion

Programming sensor networks in terms of high-level, declarative, *macroprogramming* languages is feasible and it simplifies development by abstracting away the low-level details of sensing, communication, and energy management. However, since these features are handled "under the hood" by the compiler, it becomes possible to write very inefficient programs without realizing.Future work should focus on analyzing performance and giving users a reasonable expectation as to the performance of their programs — whether through a formal cost model, or if impossible, more effective simulation of average-case scenarios. Further, we have only scratched the surface of potential optimizations for Regiment and Regiment-like languages.

We have sought to demonstrate that an approach based on functional programming and two-level semantics (monads) is more flexible and general than the relational database approach.

- The FRP framework allows reaction to events in a systematic and controlled way;
- regions can form arbitrary and powerful groupings;
- anchors and regions allow a *constructive* programming style wherein structure is built on the network rather than just data queried;
- and, finally, the (traditional) procedural abstraction mechanism allows the user

build reusable functionality.

Future work should consider the ramifications of embedding Regiment's region operators in a non-functional language (for example, regions become Java objects). We suspect such an approach would result in a greater frequency of costly synchronization with a central base-station. Further, functional programming provides a great deal of implicit parallelism [6] that Regiment is currently under-utilizing.

We have also provided an intermediate language, the Token Machine Language (TML), that provides a semantically simple and uniform target for our Regiment compiler, but is also easily human writable and readable. It is a low-overhead language — very small programs can accomplish basic data-collection tasks. It is simpler and less expressive than a language like NesC. It has no memory management and as a result lacks most of the failure modes of NesC, making it ideal for beginners.

While we have valued TML, in future work with Regiment we will aim to formulate a new intermediate language at a somewhat higher level than TML. In particular, we would like to see streaming data represented explicitly in the intermediate language. We have hopes that a common stream-processing framework can support both the macroprogramming vision as represented by Regiment as well as the declarative query processing model as seen in TinyDB.

There are definite benefits to the Regiment programming paradigm. It makes certain programs extremely easy to write. What remains is for the system to be tested "in the field" and validated for a class of sensor network applications. When that is accomplished, we will see where the future will take us. Regiment's most beneficial property in the long run will likely be its open-ended nature. It provides a platform: region processing, for which many expressive operations can be imagined. They await only a demand.

Bibliography

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Centintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, , and S. Zdonik. The design of the borealis stream processing engine. In *CIDR 2005 - Second Biennial Conference on Innovative Data Systems Research*, January 2005.

[2] T. Abdelzaher, B. Blum, Q. Cao, D. Evans, J. George, S. George, T. He, L. Luo, S. Son, R. Stoleru, J. Stankovic, and A. Wood. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks.

[3] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, J. Thomas F. Knight, R. Nagpal, E. Rauch, G. J. Sussman, and R. Weiss. Amorphous computing. *Commun. ACM*, 43(5):74–82, 2000.

[4] J. Annevelik. Database programming languages: A functional approach. In *Proc. of the ACM Conf. on Management of Data*, pages 318–327, 1991.

[5] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. *STREAM: The Stanford Data Stream Management System.* 2004.

[6] Arvind and R. Nikhil. *Implicit Parallel Programming in pH.* Morgan Kaufman, 2001.

[7] Arvind, R. S. Nikhil, D. L. Rosenband, and N. Dave. High-level Synthesis: An Essential Ingredient for Designing Complex ASICs. In *Proceedings of ICCAD'04*, San Diego, CA, 2004.

[8] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. Fad, a powerful and simple database language. In *Proc. Conf. on Very Large Data Bases (VLDB)*, 1987.

[9] R. Barr, J. Bicket, D. Dantas, B. Du, T. Kim, B. Zhou, and E. Sirer. the need for system-level support for ad hoc and sensor networks, 2002.

[10] G. E. Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-93-129, April 1993.

[11] B. Blum, P. Nagaraddi, A. Wood, T. Abdelzaher, S. Son, and J. Stankovic. An entity maintenance and connection service for sensor networks, 2003.

[12] C. Borcea, C. Intanagonwiwat, P. Kang, U. Kremer, and L. Iftode. Spatial programming using smart messages: Design and implementation. In *24th International Conference on Distributed Computing Systems (ICDCS 2004)*, March 2004.

[13] W. Butera. *Programming a Paintable Computer.* PhD thesis, MIT, 2002.

[14] D. Coore. *Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer.* PhD thesis, MIT Department of Electrical Engineering and Computer Science, February 1999.

[15] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: a compiler-controlled threaded abstract machine. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 164–175. ACM Press, 1991.

[16] R. K. Dybvig. Writing hygienic macros in scheme with syntax-case. Technical Report TR 356, 1992.

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

[18] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. *SIGPLAN Not.*, 39(4):502–514, 2004.

[19] M. Flatt. Composable and compilable macros: You want it when, 2002.

[20] B. Greenstein, E. Kohler, and D. Estrin. A sensor network application construction kit (snack). In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 69–80. ACM Press, 2004.

[21] J. Hatcliff and O. Danvy. A generic account of continuation-passing styles. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 458–471. ACM Press, 1994.

[22] W. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proc. the 5th ACM/IEEE Mobicom Conference*, August 1999.

[23] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 93–104. ACM Press, 2000.

[24] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proc. International Conference on Mobile Computing and Networking*, Aug. 2000.

[25] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* P-H, 1993.

[26] S. P. Jones and J. Hughes. Report on the programming language haskell 98., 1999.

[27] R. Kelsey, W. Clinger, and J. R. (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[28] A. Kondacs. Biologically-inspired self-assembly of 2d shapes, using global-to-local compilation. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2003.

[29] P. Levis and D. Culler. Mate: a tiny virtual machine for sensor networks. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 85–95. ACM Press, 2002.

[30] T. Liu and M. Martonosi. Impala: A middleware system for managing autonomic, parallel sensor systems, 2003.

[31] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proc. the 5th OSDI*, December 2002.

[32] E. Moggi. Computational lambda-calculus and monads. In *Proceedings 4th Annual IEEE Symp. on Logic in Computer Science, LICS'89, Pacific Grove, CA, USA, 5–8 June 1989*, pages 14–23. IEEE Computer Society Press, Washington, DC, 1989.

[33] A. Mycroft and R. Sharp. Hardware/software co-design using functional languages. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 236–251, 2001.

[34] Nagpal, Shrobe, and Bachrach. Organizing a global coordinate system from local information on an ad hoc sensor network. In *2nd International Workshop on Information Processing in Sensor Networks (IPSN '03)*, April 2003.

[35] R. Nagpal. *Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics.* PhD thesis, MIT Department of Electrical Engineering and Computer Science, June 2001.

[36] S. Nath, Y. Ke, P. B. Gibbons, B. Karp, and S. Seshan. IrisNet: An architecture for enabling sensor-enriched Internet service. Technical Report IRP-TR-03-04, Intel Research Pittsburgh, June 2003.

[37] S. L. Peyton Jones and A. L. M. Santos. A transformation-based optimiser for Haskell. volume 32, pages 3–47, 1998.

[38] Pointon, Trinder, and Loidl. The design and implementation of Glasgow Distributed Haskell. *Lecture Notes in Computer Science*, 2001.

[39] N. Priyantha, H. Balakrishnan, E. Demaine, and S. Teller. Anchor-free distributed localization in sensor networks, 2003.

[40] A. Sabry. What is a purely functional language? *Journal of Functional Programming*, 8(1):1–22, 1998.

[41] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings 1992 ACM Conf. on Lisp and Functional Programming, San Francisco, CA, USA, 22–24 June 1992*, pages 288–298. ACM Press, New York, 1992.

[42] T. Sheard. Accomplishments and research challenges in meta-programming. *Lecture Notes in Computer Science*, 2196:2–??, 2001.

[43] G. L. Steele and W. D. Hillis. Connection machine lisp: Fine grained parallel symbolic programming. pages 279–297.

[44] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997*, pages 203–217. New York: ACM, 1997.

[45] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrating communication and computation. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 430–440. ACM Press, 1998.

[46] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, Jan. 1989.

[47] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *Proc. the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.

[48] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: A neighborhood abstraction for sensor networks. In *Proc. the International Conference on Mobile Systems, Applications, and Services (MOBISYS '04)*, June 2004.

[49] B. T. G. P. S. N. with Application Specific Virtual Machines. In submission to osdi 2004.

[50] Y. Yao and J. E. Gehrke. The Cougar approach to in-network query processing in sensor networks. *ACM Sigmod Record*, 31(3), September 2002.

[51] S. Zdonik, M. Stonebraker, M. Cherniack, U. Cetintemel, M. Balazinska, and H. Balakrishnan. The aurora and medusa projects. *Bulletin of the Technical Committee on Data Engineering*, 2001.