Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2009-032 June 25, 2009

# Programming Manifolds

Jacob Beal and Jonathan Bachrach

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

# Programming Manifolds

Jacob Beal Jonathan Bachrach

Original Version: December, 2006
Edited: May, 2009

**Abstract**

Many programming domains involve the manipulation of values distributed through a manifold—examples include sensor networks, smart materials, and biofilms. This paper describes a programming semantics for manifolds based on the *amorphous medium* abstraction, which places a computational device at every point in the manifold. This abstraction enables the creation of programs that automatically scale to networks of different size and device density. This semantics is currently implemented in our language Proto and compiles for execution on Mica2 Motes and several other platforms.

## 1 Introduction

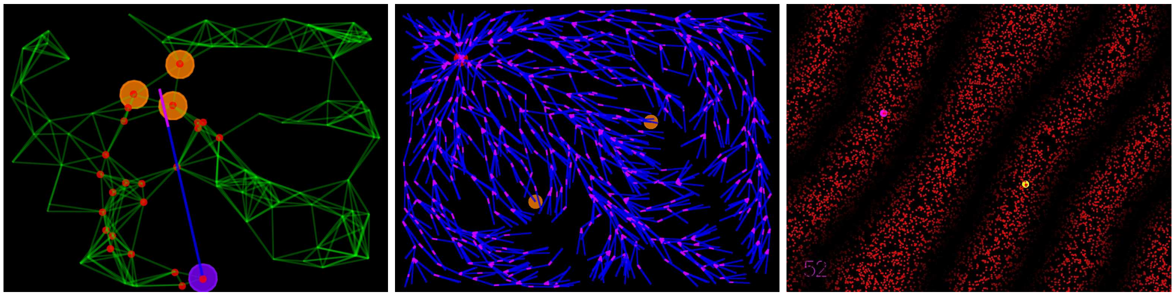
Many programming domains involve the manipulation of values distributed through a manifold. A few current examples:

- A wireless sensor network deployed throughout a large ship to allow a small crew to monitor the health of the vast collection of machinery onboard, and manage flows of water, coolant, etc. throughout the ship.[10]
- A programmable biofilm where a colony of cells coordinate chemically to capture an image or dispense a drug in carefully timed and located doses.[12]
- An ad-hoc packet radio network deployed on the roofs of buildings, doing routing and traffic management.[9]
- Reconfigurable robots coordinating to build arbitrary physical structures.[13]

Each of these are examples is a *spatial computer*—a collection of devices that fill space and whose ability to interact is strongly dependent on their proximity. The network of devices making up the computer approximates a geometric region of Euclidean space: the colony of cells covers the surface on which they grow, the sensor network fills the interior of the ship, etc.

It is natural to think about programming these spaces using geometric notions. The structure of the space, however, may be quite complicated. For example, communication on the ship may detour around bulkheads and cargo areas, while the reconfigurable robots may form an elaborate shape with complicated bends and folds. Thinking about the spaces as manifolds will make programming them easier, as we can then largely decouple a geometric description of the behavior we want from the details of communication and execution on individual devices necessary to implement it.

This paper describes a programming semantics for manifolds based on the *amorphous medium* abstraction, which places a computational device at every point in the manifold. This abstraction enables the creation of programs that automatically scale to networks of different size and device density. This semantics is currently implemented in our language Proto, compiles for execution on Mica2 Motes, and has been used to implement algorithms for sensor networks and modular robotics (Figure 1).



(a) Tracking (b) Threat Avoidance (c) Plane Wave

Figure 1: Programs for target tracking (a), threat avoidance (b), and directable plane waves (c) being verified on 100, 1000, and 10000 simulated devices, respectively.

## 2 A Brief Review of Manifolds

We will start by briefly going over what a manifold is: if you are familiar with them, feel free to skip this section.

A manifold is a space that looks like a Euclidean space locally, though its whole structure may be much more complicated, with holes and other geometric awkwardness. To be precise:

**Definition 1** *An $m$-manifold is a space* $\mathbf{M}$ *where each point in the space has a neighborhood which is homeomorphic to an open set in* $\mathbb{R}^{\mathbf{m}}$.

The $m$ is the dimension of the manifold. Since we are interested in the real world, we will mostly be interested in 2-manifolds (surfaces) and 3-manifolds (volumes).

Some examples of manifolds:

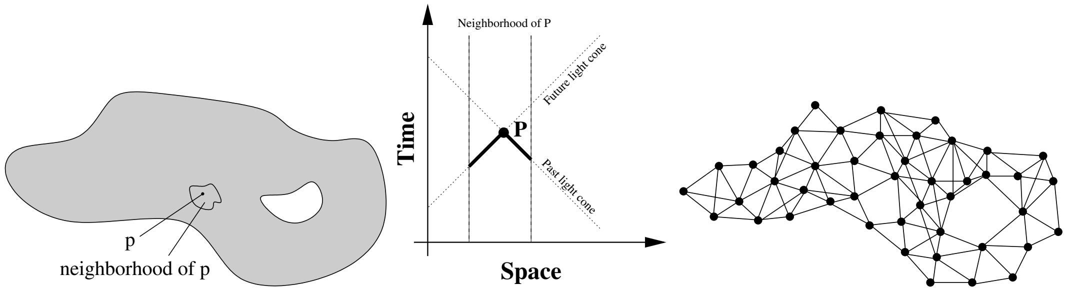
- A curve is a 1-manifold
- Figure 2(a) shows a 2-manifold. Other examples are the surface of the Earth, the road network, a Mobius strip, and a Klein bottle.
- The interior of a building and a coffee cup are both 3-manifolds

In the interests of simplicity, we will only consider manifolds which are Riemannian, smooth, and compact—that is, ones where distance and derivatives make sense, and without nasty topological quirks. This excludes a great number of interesting mathematical objects, but not much that we can build in the real world. Our semantics might apply to manifolds without these properties as well, but we have not been motivated to investigate the matter.

Functions can be defined on manifolds, mapping each point in the manifold to a value. We will call these functions **fields** after the fashion of physics, and denote a field either $\mathbb{F}$ for a generic field, or by its range (e.g. $\mathbb{R}$ for a field of numbers).

## 3 The Amorphous Medium

An *amorphous medium* is a manifold where every point is a computational device. Nearby devices share state—each device has some neighborhood in which it can access the state of other devices. Information propagates across the manifold at a maximum velocity $c$, so the processor accesses the past light cone of its neighborhood, not the current values (Figure 2(b)).




(a) Amorphous Medium (b) Information Propagation (c) Discrete Network

Figure 2: An amorphous medium is a manifold where every point is a computational device that shares state with a neighborhood of other nearby devices. Information propagates through an amorphous medium at a fixed rate, so each processor has access only to values in the intersection of its neighborhood and past light cone. An amorphous medium may be approximated by a mesh-like discrete network.

All of the infinitely many devices execute the same program, but their executions diverge due to differences in sensor values, randomness, and interaction with their neighborhoods.

Obviously, we cannot build an amorphous medium—there are infinitely many processors! We can, however, approximate it on a discrete network. Therein lies the power of the abstraction: by writing a program for an abstract continuous space, we end up with code that can be run approximately on a wide range of different real networks.

Because all interactions in the amorphous medium are local, if we can build an approximate implementation of a neighborhood, then we can approximate any computation on the amorphous medium. Fortunately, although a manifold may have a complex shape, it is defined to be simple in the neighborhood of each point.

For unit disc communication, for example, the neighborhood can be approximated as the set of 1-hop neighbors in the network, with $c$ equal to the average hop distance divided by the time between periodic updates.

Note also that there is a duality between the amorphous medium and the network used to approximate it. Given an amorphous medium, we can evaluate how well various networks approximate it; given a network, we can find an amorphous medium which it approximates.

We will not treat the transformation between continuous and discrete in detail in this paper: see [3] and [6] for more information on this issue.

# 4 Evaluation Model

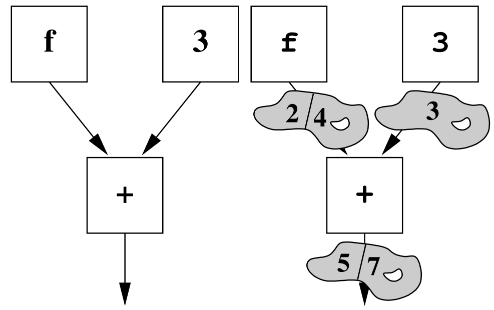
Our language, Proto, uses ideas from both functional and stream programming to provide an evaluation model for computing with an amorphous medium.

Programs written in Proto can be thought of either as a network of streams or as a dataflow graph evaluated repeatedly over time. In this description, we will use the dataflow graph interpretation.

There are two basic elements in the language:

**Definition 2** *An expression is a function* $e : \mathbf{M} \rightarrow \mathbb{F}^{\mathbf{i}}$ *that takes an amorphous medium as input and produces a set of fields as output.*

**Definition 3** *An operator is a function* $o : \mathbb{F}^{\mathbf{i}} \rightarrow \mathbb{F}^{\mathbf{j}}$ *that takes a set of fields as input and produces a set of fields as output.*




(a) Expression (b) Evaluation

Figure 3: An expression is evaluated at a point in time by applying it to an amorphous medium, producing a set of fields as output. For example, the expression **(+ f 3)** produces a field where every point mapped to $n$ by **f** maps to $n + 3$ in the new field. General evaluation is instantaneous evaluation repeated at fixed intervals.

For example, the expression **4** takes an amorphous medium and produces a field where every point in the amorphous medium maps to $4$, and the operator **sqrt** takes a scalar field and produces a scalar field where every point that mapped to $n$ in the input field maps to $\sqrt{n}$ in the output field.

The means of composition is implied by the functional definition. We will use LISP notation to show composition, though the reader should remain aware that Proto is not a LISP.

Composing an operator with a compatible expression produces another expression. For example: **(sqrt 4)** is an expression that takes an amorphous medium and produces a field where every point in the amorphous medium maps to 2.

Composing an operator with another operator produces an operator. An expression can be converted into an operator by means of a special scope function $s_i : \mathbb{F}^{\mathbf{i}} \rightarrow \mathbf{M}$, which takes a set of fields and returns an amorphous medium which is the intersection of their domains. Composing the scope function with an expression $e$ produces an operator $s_i e$ that gives the value of an expression on the amorphous medium where the input variables are defined. Using this, we can define compound operators that include arbitrary expressions: for example, **(lambda (x) (* 2 x))** is an operator that takes a scalar field and produces a scalar field where every point that mapped to $n$ in the input field maps to a $2n$ in the output field.

We can now define the instantaneous evaluation of an expression—its evaluation at a given point in time—as simply its application to an amorphous medium, producing a set of fields (Figure 3). To find out what values have actually been computed, we must further evaluate the field at particular points.

To evaluate an expression across time, we perform a sequence of instantaneous evaluations at fixed intervals.[1] Each instantaneous evaluation depends only on the previous evaluation (via the **delay** operator—see Section 9). In the future, we look to allow make evaluation continuous in time instead of at discrete fixed intervals.

# 5 Instantaneous Pointwise Computation

We now know enough to translate any purely functional operation on a normal computer into a pointwise operation on an amorphous medium: the amorphous medium version simply applies the operation uniformly to every point in the amorphous medium. A few examples:

[1] In the approximate implementation, we will not attempt close time synchronization. It is sufficient to have a low skew in evaluation rate.

- $* : \mathbb{R}^{\mathbf{i}} \rightarrow \mathbb{R}$ takes any number of real-valued fields and returns a real-valued field where each point maps to the product of the values that it mapped to in the input fields.

- $truncate : \mathbb{R} \rightarrow \mathbb{R} \times \mathbb{R}$ takes a real-valued field and returns two real-valued fields: the first containing the integer part of values in the input field, the second containing the fractional part.

- $tup : \mathbb{F}^{\mathbf{i}} \rightarrow \mathbb{T}$ takes any number of fields and returns a field of tuples. *1st* : $\mathbb{T} \rightarrow \mathbb{F}$ takes a field of tuples and returns a field of the first component of each tuple, *2nd* : $\mathbb{T} \rightarrow \mathbb{F}$ returns the second, and so on.

Allowing operator-valued fields, we have pointwise first order functions as well. Nor is there any block to defining recursive operators.

We can now describe any instantaneous pointwise computation on an amorphous medium—that is, any computation where the results at a point do not depend on its history or the history of values at other points.

A few issues in pointwise operations warrant specific attention:

- Purely functional exception handlers can be translated for use on an amorphous medium. For example, by adjoining a special **error** value to every operator, we can allow composition of operators without extremely strict compatibility checking: any input not handled properly by the operator results in an output of **error**.

- A pointwise random number generator is not particularly useful, since there are so many points that the resulting field is completely homogeneous. Instead, we will use a **random** operator that creates an arbitrary finite partition of the amorphous medium, then assigns a single value to each element of the partition.

- For pointwise expressions, conditionals can operate pointwise. When side-effects or computations across the space are involved, we will need two varieties of conditional (see Section 8).

- Input can be gathered via a pointwise **sense** operation that takes an identifier for an input device and returns its current value. Output can be produced similarly with a pointwise **act** operation that takes a value and supplies it to an output device. The output is a side-effect, however, and thus may interact with conditionals.

# 6 Motivating Examples

As we begin to discuss more complex operations, we will carry along two motivating examples, **gradient** and **bound**:

```
(def gradient (src)
  (letfed ((n infinity (mux src 0 (min-hood (+ (nbr n) (nbr-range))))))
    n))

(def bound (src lim boundary)
  (if boundary 0 (<= (gradient src) lim)))
```

For now, do not worry about understanding the code: it is enough to know that the **gradient** function measures the distance from every point to a source by setting the distance at the source to zero, then relaxing across the whole space using the triangle inequality, and the **bound** function builds on top of this, selecting the interior of a region by clipping against a boundary.

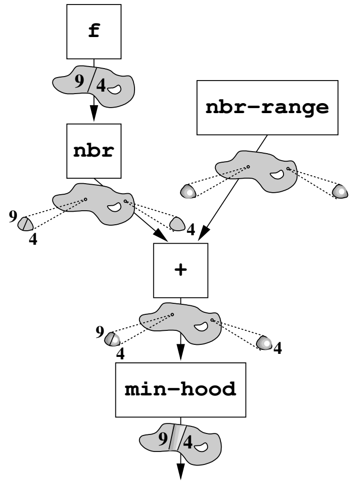



Figure 4: The **nbr** operation selects neighborhood values for computation at each point. Computing is carried out on neighborhoods with pointwise operations transformed to apply to neighborhood fields. Finally, a summary operation collapses each neighborhood of values into a single value for each point.

# 7 Computation Over Neighborhoods

Now we will begin to extend our semantics to take advantage of the amorphous medium, beginning with computations that extend across local areas of the manifold. To do this, we're going to use something new: a field-valued field.

The operation **nbr** takes a field as input and returns a field of fields, where each point maps to a field where the domain is the neighborhood and the values are the values in its past light cone (Figure 4). Included in the neighborhood field for each point is the value of the point itself at the current time. Note that including this value changes little, since there will generally also be neighbors arbitrarily close by which map to arbitrarily close values.

Notice that **nbr** isn't so much performing a computation as selecting the values that will be used for computation. In order to produce a usable output, we'll have to summarize this field of values back down into a single value in the end.

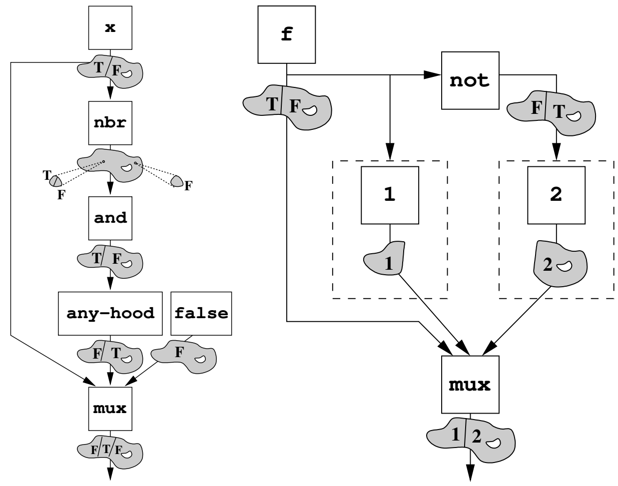
We will sometimes also need information about the structure of the manifold, so we also define special expressions for geometric information: **nbr-range** and **nbr-angle** give fields of spatial displacement to neighbors, **nbr-density** gives a field of the density of neighbors, and **nbr-lag** gives a field of time displacement to neighbors.

Polymorphic forms of pointwise operations can safely be applied to these fields of neighborhood values, "one level down." Scalar values can be mixed into the computation as well—the polymorphic form simply uses the scalar in operating on each point in the neighborhood.

Finally, the field of neighborhood fields is summarized back into an ordinary field with one of several summary operators. At present, we have defined and used five summary operators: **int-hood** (which takes an integral), **any-hood**, **all-hood**, **min-hood** and **max-hood**. [2] We have not yet determined whether these summary operators are sufficient; if not, then the set of operators can simply be extended.

For example, **gradient** includes an expression applying the triangle inequality, **(min-hood (+ (nbr n) (nbr-range)))** (Figure 4). This selects two neighborhood fields—one filled with estimates of distance to source, the other with the distance to neighbors—and adds them to find the estimates distance to the source

[2] The **min-hood** and **max-hood** operations are actually infimum and supremum limits, ensuring that their value is always well defined.




(a) Example of **mux** (b) Example of **if**

Figure 5: The conditional **mux** runs both computations and uses the test to select an answer, allowing the whole space to influence both computations. For example, **(mux x (not (all-hood (nbr x))) false)** finds the boundary of region **x** by selecting the points inside **x** from the group of points with a neighbor outside of **x**. The conditional **if**, on the other hand, is a syntactic operator which branches computation by restricting the domain of the fields for each branch, then combines the results using **mux**. Shown above is a simple example **(if f 1 2)**.

through each neighbor. The triangle inequality is then applied by taking the minimum of all these distances with **min-hood**.

When approximated on a network, the **nbr** operation implies communication: devices proactively broadcast the values that will be needed by their neighbors. In some cases, static analysis can reduce or eliminate communication costs: for example, the **nbr** of a constant need not be broadcast. Transformed pointwise operations are then performed on the collection of values most recently broadcast. Finally, the five summary functions above can all be approximated well with finite samples.[3]

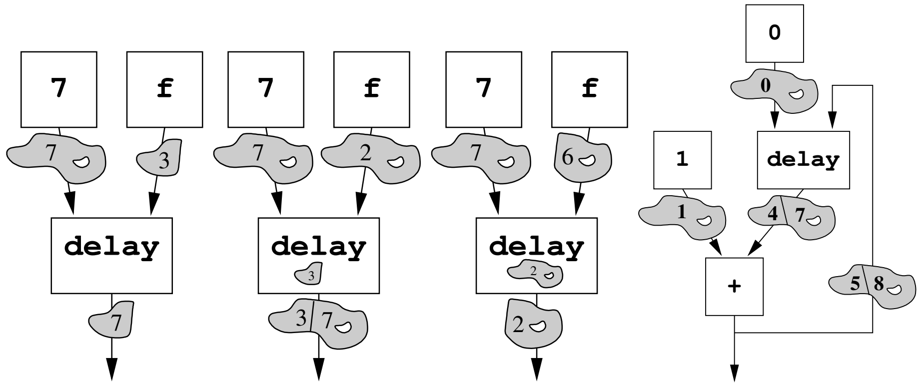
We now have a mechanism for computations that extend through neighborhoods. Computation on general regions of the manifold awaits two more ingredients: conditionals, which will allow us to specify the region, and state, which will allow information to travel across long distances.

# 8 Conditional Computation

In any model of computation, conditionals are interpreted either as branching or selection. In the branching interpretation, a test is evaluated and the result determines which of two computations is executed. In the selection interpretation, both computations are performed, and the test determines which result is returned. The branching interpretation is most common, since it makes side-effects easy to understand and control.

In computing on an amorphous medium, however, we will find that we need to use both approaches. First, we will assume that the test produces a field of booleans, and the value of the field at each point determines which computation should produce the value for that point. The neighborhood of a point, however, may contain points where the test produces an opposite value. The question, then, is this: if the conditional computation contains neighborhood operations, should they be able to use the values of neighbors where

[3]A few subtleties of the approximation: **int-hood** is defined as a Lebesgue integral, **all-hood** is defined to yield true when the set of false neighbors has measure zero, and **any-hood** yields false when the set of true neighbors has measure zero.




(a) Time T-1 (b) Time T (c) Time T+1 (d) Feedback Loop

Figure 6: The **delay** operator, **(delay default input)**, takes its scope from the current input and its values from the previous input, filling in any missing values with the initial value. The figure above shows three successive time-steps of the evaluation of **(delay 7 f)**. The **delay** operator is used to create state variables using a feedback loop. For example, the counter shown in (d) starts at zero and increases by one each round.

the test goes the other way?

Under the branching interpretation, which we will name **if**, the test is used to restrict the space involved in the computation, and neighbors in the opposite branch are excluded. The branching interpretation is useful for preventing unnecessary computation and for controlling actuation, which has side-effects.

For example, **bound** uses **if** because it needs the boundary to stop the gradient from propagating further.

Under the selection interpretation, which we will name **mux** for its multiplexing behavior, the space is not restricted. This interpretation is useful for computations that span space. For example, an impermeable boundary for a region identified with a boolean field **x** can be calculated using the expression **(mux x (not (all-hood (nbr x))) false)**, which will return true for the points inside the region which have a point outside as a neighbor (Figure 5(a)). If we substitute **if** for **mux**, then the computation will fail, because no point outside the region is accessible to the neighborhood operation in the true branch.

For example, **gradient** uses **mux** because the devices near the source need to be able to see the source in order to know how far they are from it.

The implementation of **mux** is easy: it is a pointwise operator **(mux test true-expr false-expr)** that just uses the value of the first field to choose between values from the second and third fields.

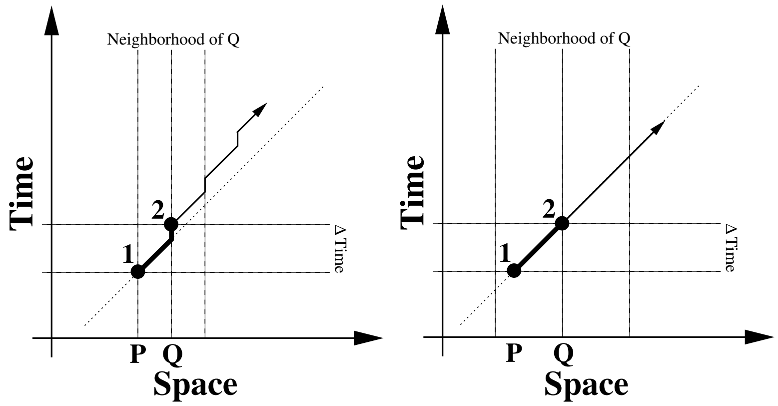
We can then implement **if** using **mux** and a restriction of the domain. We annotate such a restricion in our diagrams as a dotted box with a boolean **test** field to it, where the domain inside the box contains only those points where the **test** field maps to true. We will not allow the programmer to specify these restrictions directly, however, as that introduces the danger of ending up with undefined values.

Putting these pieces together, we can define **if** as a syntactic operator **(if test then else)** which wraps the terminals of the **then** and **else** expressions in a **restrict** operation that implements this change of domain, then pastes the results together using **mux** (Figure 5(b)). Using **test** for both splitting the branches and pasting together their outputs guarantees that nothing in the domain is left undefined.

We can then implement any other conditional operator we want on the basis of **mux** and **if**.

## 9 Computation With State

We now have computations that extend over small distances in space, and the ability to restrict the space in which they apply. In order to extend them across long distances in space, we will need to add state to our




(a) Insufficient Time Resolution (b) Smooth Propagation

Figure 7: Discrete time evaluation causes "hiccups" in long-distance information propagation when the delay between evaluations is longer than the time it takes information to propagate across a neighborhood. When Q operates on its neighborhood at point 2, the most distant accessible value is that calculated by P at point 1. If space is the limiting factor (a), then there is an additional delay between when information from P arrives and when Q can do its next calculation. If time is the limiting factor (b), then Q can repropagate the values from P with no apparent delay.

computations. Incidentally, this also gives us the tools for computations that extend over time.

In keeping with our functional approach, we establish state with a feedback loop: a state variable is defined by an initial value and an update function which uses values from each time-step to calculate the values for the next.

Fundamental to this is the **delay** operator, **(delay init value)**, which takes its domain from the current time-step and its values from the previous time-step (Figure 6). At startup and following space restriction, there may be undefined values in the domain: these are filled in from **init**.

Using delay, we can set up a feedback loop where the output of the delay operator is fed into an expression that computes the next input. For example, we can make a counter by making a loop where we add one to the delayed value (Figure 6(d)).

Another common use of feedback is long-distance communication. The **gradient** function uses it in this manner, chaining relaxation with a feedback loop incorporating a neighborhood operation. The feedback loop provides a slot for the communication to chain through.

This structure is captured with a syntactic operator, **(letfed ((name init expr) . . . ) body)**, which establishes a set of state variables that can use each other's old values during updates. The body of the **letfed** expression can then calculate using the values of the feedback variables.

## 10 Spatial Computation

We now have all of the tools necessary to create general computations that extend over arbitrary portions of the manifold. The general form of such a computation is local propagation using neighborhood operations, chained by means of a set of feedback variables holding the values to be communicated.

As long as evaluations happen frequently enough, this combination of local propagation and feedback variables can propagate information across arbitrary distances at the maximum velocity $c$. If the delay between evaluations is shorter than the time it takes information to propagate across a neighborhood, then every evaluation will happen just as some point is receiving the information, and it can be propagated instantaneously. If the delay is longer, however, then long-distance propagation will "hiccup" as information reaches the edge of a neighborhood, but cannot pass beyond it until the next evaluation occurs.

One pattern we have found broadly useful is relaxation, where points start with an extreme value, except for a few seeds which know their final value. The desired field is then calculated recursively, as neighbors of the seeds relax toward their final value, which allows their neighbors to relax as well, and so on, spreading outward across the manifold.

The **gradient** function is an example of code that uses this pattern. Repeating the code from above:

```
(def gradient (src)
  (letfed ((n infinity (mux src 0 (min-hood (+ (nbr n) (nbr-range))))))
    n))
```

Points outside the source start at infinity, while points within start at zero. Every point looks at its neighborhood and calculates the shortest distance to the source through each neighbor, then selects the minimum as its own distance. The net effect is that as time progresses, correct distance values spread outward from the source, until the desired distance field has been calculated for the entire manifold.[4]

Given this set of building blocks, it is easy to express complex programs simply. Implemented in our language Proto, for example, it takes only 28 lines to describe a target tracking program and 23 lines to describe a program for threat avoidance (Figure 1)—see the Appendix for the code, and [3] for a line-by-line explanation of these examples.

The manifold semantics we have presented allows the same code to run on networks with widely varying numbers and distributions of nodes. Figure 8 shows target tracking scaling across networks ranging from 20 to 10,000 nodes. The figures show target detection as large orange circles, the reporting node as a large red circle, and the reported position of the target as a blue line tipped with magenta. Network connections are green (except in the 10K case, where there are too many to show), nodes where tracking data is flowing are small red circles, and other nodes are tiny red dots. All of these different scales provide the same behavior as the 100 node network in Figure 1(a) at different resolutions.

Moreover, the compiled code is compact enough to fit (along with a custom virtual machine to execute it) on Mica2 motes, which have a scant 4KB of RAM and a 16Mhz microcontroller.

Not all spatial programs can be expressed well using these primitives. Inherently discrete algorithms, such as TDMA or matrix-based optimization, can be run only awkwardly, by recreating the discrete space within the continuous space.

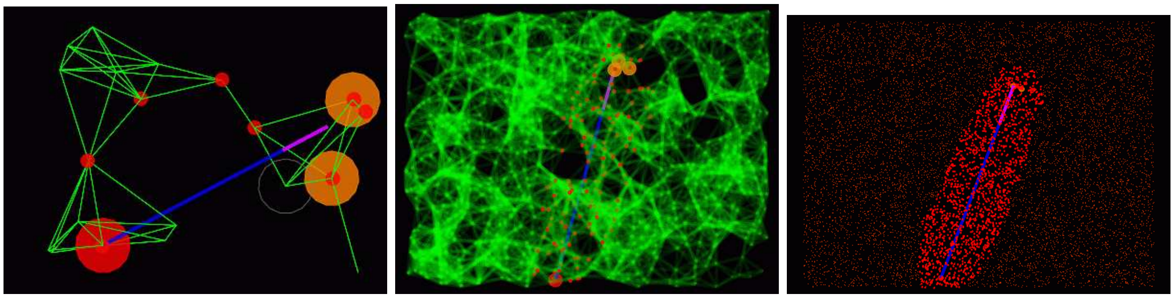
# 11 Conclusion

We thus have a programming semantics that gives us a global model of computation on an amorphous medium, which is implemented in our language Proto. This model is powerful because it allows us to prescribe the behavior of a spatially embedded network without concerning ourselves with the details of its deployment or communication patterns. Furthermore, programs can be very simple and use primitives that carry over some intuitions from ordinary single-processor programming.

## 11.1 Related Work

This work does not, of course, exist in a vacuum. Previous work on amorphous medium languages proposes the amorphous medium abstraction[4], general strategies for control[7], and an ancestor language of Proto[5]. Recently, we described[6] how the abstraction simplifies engineering of emergent behavior and investigated its applicability to sensor network programming[3]

[4]Note that this code does not handle a changing source, since it cannot rise in response to the source getting farther away from a point. That version of **gradient** is significantly more complicated.



(a) 20 devices (b) 1,000 devices (c) 10,000 devices

Figure 8: Target tracking scales across a wide range of networks. Shown above are simulations of with twenty, one thousand, and ten thousand devices, all providing the same behavior as the 100 node network in Figure 1(a) at different resolutions. (Note: the 10K simulation does not show network connections in green, as there are too many)

Others have envisioned computing on platforms like the amorphous medium. MacLennan's field transformation computers[18] use a different basis for computation that does not require locality in its operations. The Continuum Computer Architecture[20] envisions building a real platform that closely approximates an amorphous medium, though with a different programming model. The Connection Machine[14] provided a grid of processing, which *Lisp[16] allowed users to manipulate in terms of fields, but the programming model was firmly wedded to the discrete grid structure of the implementing hardware.

Cellular automata computing has proved a useful way of approximating the behavior of continuous media[21], and there is in fact a continuous formulation of CAs.[17] CAs could certainly be used as a spatial computer on which Proto could be executed, though the regularity might create problems for approximation. At the present time, we are not aware of a language for CAs which supports scaling to machines of different resolution.

The specification of behavior in a CA is inherently local, however, and lends itself to investigation of emergent behavior rather than engineered computation.

The language Regiment[19] is a sensor-network language which operates on geometric regions of space, but it is focused on data-gathering and only distributes some operations across space.

Other work on languages in amorphous computing [1] has shared the same general goals, but has been directed more towards problems of morphogenesis and pattern formation than general computation. A notable exception is Butera's work on paintable computing[8], which allows general computation, but lacks an abstraction barrier separating an applications programmer from low-level network details.

Finally, the structure of Proto as a dynamic network of streams is strongly influenced by Bachrach's previous work on Gooze[2], as are many of the compilation strategies used to compact Proto code for execution on Motes. There is a long tradition of stream processing in programming languages. The closest and most recent work is Functional Reactive Programming (FRP) [11] that is based on Haskell [15], which is a statically typed programming language with lazy evaluation semantics. In these systems, less attention is spent on runtime space and time efficiency, and the type system is firmly wedded to Haskell, with all of its strengths and weaknesses.

### 11.2 Future Directions

The work described herein only begins to answer the host of interesting questions about how to program on manifolds. From our perspective, the most pressing open problems are:

- How can the evaluation model be extended to continuous time without losing the advantages of discrete-time programming models?

- How can approximation error be usefully characterized and bounded?
- How can relaxation be modified to allow non-monotonic results?
- How can the behavior of a program on a changing manifold be usefully characterized?
- How can actuation that reshapes the manifold be usefully described and controlled?

As always, a strong driver for future investigation will be application of these ideas to new problems, which will best reveal their weaknesses.

## References


[1] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homs y, T. Knight, R. Nagpal, E. Rauch, G. Sussman, and R. Weiss. Amorphous computing. Technical Report AIM-1665, MIT, 1999.

[2] Jonathan Bachrach. Gooze: a stream processing language. In *Lightweight Languages 2004*, November 2004.

[3] Jonathan Bachrach and Jacob Beal. Programming a sensor network as an amorphous medium. In *Distributed Computing in Sensor Systems (DCOSS) 2006 Poster*, June 2006.

[4] Jacob Beal. Programming an amorphous computational medium. In *Unconventional Programming Paradigms International Workshop*, September 2004.

[5] Jacob Beal. Amorphous medium language. In *Large-Scale Multi-Agent Systems Workshop (LSMAS)*. Held in Conjunction with AAMAS-05, 2005.

[6] Jacob Beal and Jonathan Bachrach. Infrastructure for engineered emergence in sensor/actuator networks. *IEEE Intelligent Systems*, pages 10–19, March/April 2006.

[7] Jacob Beal and Gerald Sussman. Biologically-inspired robust spatial programming. Technical Report AI Memo 2005-001, MIT, January 2005.

[8] William Butera. *Programming a Paintable Computer*. PhD thesis, MIT, 2002.

[9] Benjamin Chambers. The grid roofnet: a rooftop ad hoc wireless network. Master's thesis, MIT, 2002.

[10] Fred Discenzo, Francisco Maturana, and Raymond Staron. Distributed diagnostics and dynamic reconfiguration using autonomous agents. In *International Conference on Complex Systems 2006*, June 2006.

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

[12] Drew Endy. Foundations for engineering biology. *Nature*, 438:449–453, November 2005.

[13] S.C. Goldstein, J.D. Campbell, and T.C. Mowry. Programmable matter. *Computer*, 38(6):99–101, June 2005.

[14] W.D. Hillis. *The Connection Machine*. MIT Press, 1985.

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

[16] C. Lasser, J.P. Massar, J. Miney, and L. Dayton. *Starlisp Reference Manual*. Thinking Machines Corporation, 1988.

[17] Bruce MacLennan. Continuous spatial automata. Technical Report Department of Computer Science Technical Report CS-90-121, University of Tennessee, Knoxville, 1990.

[18] Bruce MacLennan. Field computation: A theoretical framework for massively parallel analog computation, parts i-iv. Technical Report Department of Computer Science Technical Report CS-90-100, University of Tennessee, Knoxville, February 1990.

[19] Ryan Newton and Matt Welsh. Region streams: Functional macroprogramming for sensor networks. In *First International Workshop on Data Management for Sensor Networ ks (DMSN)*, August 2004.

[20] Thomas Sterling and Maciej Brodowicz. Continuum computer architecture for nano-scale and ultra-high clock rate technologies. In *International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, January 2005.

[21] Tommaso Toffoli and Norman Margolus. *Cellular Automata Machines: A new environment for modeling*. MIT Press, 1987.

# A Example Code

Note: the code below uses an optional extra argument for **min-hood** and **max-hood** that tells which element to use for comparison.

## A.1 Target Tracking

In this code, a clique of nodes detecting a target estimate its location by averaging their coordinates. A channel is created along the shortest path to the monitoring station (**dst**), and the location estimate flows back along that channel, so that the information need not be transmitted to uninvolved portions of the network.

```
(def local-average (v)
  (/ (int-hood v) (int-hood 1)))

(def gradient (src)
  (letfed ((n infinity
              (+ 1 (mux src 0 (min-hood (+ (nbr n) (nbr-range)))))))
    (- n 1)))

(def grad-value (src v) ; flow values down gradient
  (let ((d (gradient src)))
    (letfed ((x 0 (mux src v
                     (2nd (min-hood (nbr (tup d x)) 1st)))))
      x)))

(def distance (p1 p2)
  (grad-value p1 (gradient p2)))

(def dilate (src n) (<= (gradient src) n))

(def channel (src dst width)
  (let* ((d (distance src dst))
         (trail (<= (+ (gradient src) (gradient dst)) d)))
    (dilate width trail)))

(def track (target dst coord)
  (let ((point
         (if (channel target dst 10)
           (grad-value target
             (mux target
                  (tup (local-average (1st coord))
                       (local-average (2nd coord)))
                  (tup 0 0)))
           (tup 0 0))))
    (mux dst (- point coord) (tup 0 0))))
```

## A.2 Threat Avoidance

Given coordinates, a threat sensor and a model of exponentially decaying threat, we the expected safest path to a destination is calculated by relaxation and gradient descent.

```
(def exp-gradient (src d)
  (letfed ((n src (max (* d (max-hood (nbr n))) src)))
    n))
```

```
(def sq (x) (* x x))

(def dist (p1 p2)
  (sqrt (+ (sq (- (1st p1) (1st p2)))
           (sq (- (2nd p1) (2nd p2))))))

(def l-int (p1 v1 p2 v2) ; approx. line integral
  (pow (/ (- 2 (+ v1 v2)) 2) (+ 1 (dist p1 p2))))

(def max-survival (dst v p)
  (letfed
      ((ps 0 (mux dst 1
               (max-hood
                (* (l-int (nbr p) (nbr v) p v) (nbr ps))))))
    ps))

(def greedy-ascent (v coord)
  (- (2nd (max-hood (nbr (tup v coord)) 1st)) coord))

(def avoid-threats (dst coords)
  (greedy-ascent
   (max-survival
    dst
    (exp-gradient (sense :threat) 0.8) coords) coords))
```