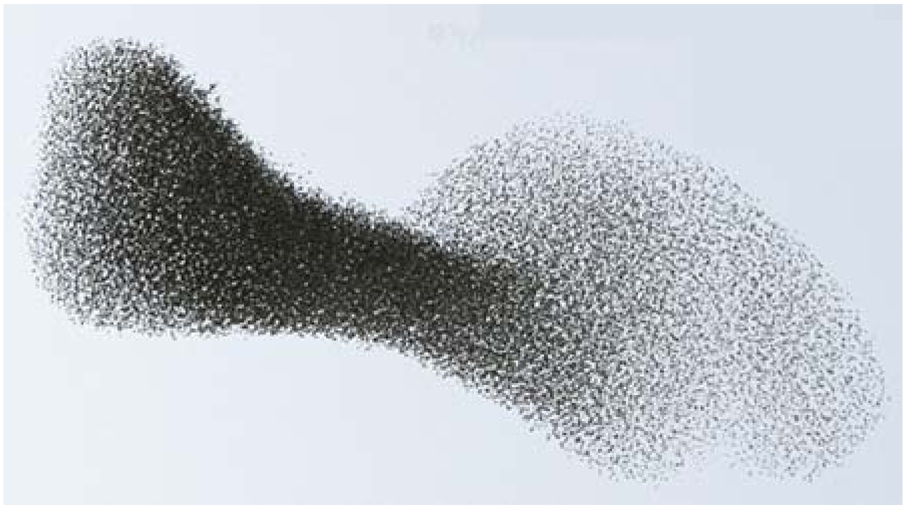
Swarm



A Cellular Programming Language

Greg Bramble gmb2106@columbia.edu
Thomas Chau tc2165@columbia.edu
Jason Gluckman jbg2113@columbia.edu
Rajesh Ramakrishnan rr2318@columbia.edu

Programming Languages and Translators, Fall 2007: Prof. Stephen Edwards.

Abstract

Swarm is a functional dataflow language aimed at building simulations. Its programs are represented by directed graphs of cells. Cells, the basic functional unit, perform computation by responding to particular sets of stimuli and propagate reactions to their neighbors. To achieve a more intricate machine, a programmer must compose primitive cells into more complex ones, which in turn may be composed into yet larger units. This proposal will describe the important features, explain this model of computation, and provide some snippets of code.

Motivation

Everything in the world behaves in a cellular manner, participating in vast interactions in which units feel each others' influences and emit their own. For example, by universally speaking the language of electromagnetism, atoms compose many interesting molecules. These complexes then compose themselves into larger structures, giving rise to cells. The cells, on the next order of magnitude, act in concert and unite themselves by circulatory networks. Finally, humans, too, interact in a broad social framework and construct organizations to handle mass action. While much reductionist work has been done to probe basic properties of these systems' constituents, important applications in biology and social science rely on understanding aggregate behavior in general. Fortunately, the current trends in hardware are beginning to meet the needs of such systems where complexity arises from the synergy of many parallel elements.

The days of the single processor are becoming history. They were characterized by programs which were essentially lists of instructions to be processed sequentially. Nearly all high-level languages in existence were written for and conceptually constrained by this paradigm. Even object-oriented programming, whose idea was to group instructions as more distinct entities, is firmly implanted in the original paradigm. While objects maintain their own states, they are nonetheless are filled with functions that must execute and return before the "main" function can continue. The OO paradigm more resembles a guideline for how to organize folders than a break from the procedural style that preceded it. As a consequence, we face a software crisis in which traditional languages have failed to capture parallel computation. We ask ourselves: how can we break away and write code to exploit new multiple processor technology?

The goal of Swarm is to allow a programmer to *explore emergent phenomena* conveniently. But even more importantly, Swarm will aim to establish a platform on which a *programmer can write code that parallelizes naturally*.

Data Types

The most important concept is that of the cell. It is a functional unit that may be bound to other units and responds to environmental stimuli by performing an internal computation and then secreting a response. The structure of all programs in Swarm takes the form of directed graphs. These graphs may be manipulated by cells themselves, which create, destroy, bind, and unbind each other. Primitive types such as numbers and text are special cells which propagate the value of their data to their neighbors.

Computational Model and Its Programming Features

Cells have facets and defined binding sites – but while these facets define interfaces between units, they are totally unlike Java Interfaces in that the possible ways to stimulate the facet’s binding sites are not enforced by the compiler; in addition, the facets may be stimulated by or observed by arbitrarily many nearby cells. Values are propagated automatically.

Binding between facets is not checked at compilation: there are no strict bond types. However, the cell membrane’s binding sites *can* filter the stimuli they receive, deciding whether to permit data inside the cell based on the data’s value, properties, or type. The cells are only stimulated to compute when the proper set of stimuli impinge on their binding sites.

Although the programming style is non-imperative and oriented around the propagation of data, and there are no assignment statements, it is still possible to have state – cells can accept values and propagate the values back to themselves. By doing this, you can persist the value until a new one is provided. (This tactic of binding cells to themselves allows for recursion).

Functionality

Swarm will allow users to make new cells by declaring binding sites and defining the innards as compositions of more primitive ones by writing the structure (e.g. a directed graph that specifies how various cell facets feed or receive inputs and outputs from each other). The net output of the cell is then whatever is allowed to diffuse out of the membrane from the underlying structure. To aid in construction, we will supply several fundamental cells:

- `BIND`: binds two given cells’ facets
- `UNBIND`: unbinds two given cells
- `CREATE`: given the encoding of a lattice, constructs its corresponding cell.
- `EXTERNALIZE`: responds to the encoding of a facet, an input cell, and a target cell; it attempts to bind the input cell to the chosen facet of the target cell. However, the target cell may not want multiple cells bound to that facet, in which case it could excrete the Externalizer into a neighbor. This could produce data structures and is also a way to traverse networks of cells.

Swarm will also supply cells for standard and graphical I/O; conditional control flow; basic mathematics and string process; and other core features as feasible in the given timeframe.

Syntax

```
Hypotenuse [(.A, .B) -> SQUARE -> ADD -> SQRT -> .OUT]
Test [(INT:3, INT:4) => (.A, .B) Hypotenuse.OUT ->STDOUT ]
```

This is a simple example of a cell that computes the hypotenuse of a triangle, given the lengths of two legs. We define the `Hypotenuse` cell and encode its structure between square brackets. Each of the facets [which represent binding sites for the cell, allowing for input and output] are preceded by dots (“`.`”). In this example, the values of A and B are passed to the `SQUARE` cell. The output (a^2, b^2) is passed to the `ADD` cell, which adds to yield (a^2 + b^2). This is passed to the `SQRT` cell which then yields sqrt(a^2 + b^2), which is then passed to the `.OUT` facet. The test program passes the pair (3,4) directly to the facets `(.A, .B)` to the `Hypotenuse` cell, which is then directed to `STDOUT` (which sends data to the console)

The syntax is freeform. The language is not whitespace-sensitive! Definitions that require multiple lines are delimited with semicolons. Labels preceded by dots (“`.`”) are recognized as facets of either the cell that contextualizes them, or the cell that they are used in. Arrows (`->`) indicate directed edges from facet to facet. Facets may be omitted for cells with one or variable-number input/output facets. In the example above, `ADD` takes in a variable number of inputs and returns to one output facet, reducing the dimension of the input by 1 [e.g. matrix → vector]. `SQRT` performs square root on each element, preserving the dimension.

Facets

In this notation for defining graphs, facets and edges may be distributed over groups delimited by parentheses. For example, `(.A, .B) -> .G (Blah)` would bind both .A and .B to the .G facet of the `Blah` cell. `(.A, .B) -> (.G, .E) (Blah)` would have a cross-product effect, pairing A with G and E, and B with G and E. Situations that call for element-wise coupling use the `=>` operator. `(.A, .B) => (.G, .E)(Blah)` would specifically bind A to G and B to E.

We can contextualize facets both in prefix form:

```
.C(Blah)
```

and postfix

```
Blah.C
```

so facets may be used as binders and bindees under (`->`), allowing for “chains”:

```
A.SampleCell -> .C(Blah)(.D, .E) => (.F, .G)(Foo)
```

When no facets are specified, the default behavior is to bind all inputs to all outputs.

Defining, Instancing and Labeled Cells

Cells are defined using the square bracket notation:

```
NAME_OF_CELL [<sequence of statements>]
```

Each statement is a chain that binds facets together:

```
(MyCell:A).out -> .C(StaticCel)(.D,.E) => (.F,.G)(Foo);
```

Cells can be labeled through the colon syntax so that later references are known to point at the same object, such as in

```
/* MyCell : A, Hypotenuse : B, MyCell : F
   MyCell has .in input and .out output */
A.out -> in.(MyCell : D);
     (out.F, out.B) -> Hypotenuse -> A;
```

In this segment, A, D, and F are distinct instances of MyCell. Both A's are the same, so that the output is fed back into the cell. The Hypotenuse cell is not explicitly instantiated.

Anonymous Bind Cells

Anonymous bind cells may be declared and stimulated within an enclosing cell definition. When they are stimulated, they temporarily alter the enclosing cell's bindings according to the lattice defined within the unnamed braces.

```
NAMED_CELL [.FACET_A -> [.FACET_B-> SQRT -> .OUTPUT]]
```

In this example, a stimulation to .FACET_A of NAMED_CELL would cause NAMED_CELL's .FACET_B to bond to a SQRT cell which outputs to NAMED_CELL's output. Anonymous bind cells cannot have their own set of named facets and are assumed to have this single special behavior of rebinding parts of their enclosing cell.

Comments, Literals and Data Typecasting

Comments are C-style. Literal values are constructed through the primitive cell-type cells. For these, their instance label is equivalent to their actual value:

```
INT : 144 -> SQRT -> STDOUT; /* sqrt(144) = 12 */
FLOAT : “12.41”               /* quotes around value */
STR : “Blah”                  /* String*/
```

The static versions of the cells act as data typecasting cells. Using `C printf` syntax, a string cell can take input and populate appropriate fields:

```
FLOAT:“12.41” -> INT -> STDOUT            /* 12 */
FLOAT:“12.41” -> STR:“Cost: %f” -> STDOUT /* Cost: 12 */
```

Building Structures

Example 1: named instantiation, excretion, and binding

Cell lattices can be grown as cells, manufacture other cells, and bind them. For example, this simple cell, when stimulated, reproduces another cell of the same type, pushes it outside, and binds to it.

Visually:

```
   O       becomes       O--O
R [ .trigger -> [ (R:newInstance).west >> .east]]
```

An instance of the R cell, when triggered, will manufacture a new instance whose .west facet binds to the originator's .east. The `>>` operator indicates that the new cell must be 'pushed out' to the exterior and then bound.

Example 2: dynamic instantiation

It is possible to dynamically build a cell, given the cell definition:

```
Factory[ .cell_definition -> NEW -> .whatever_output ]

Test [  B[(.a,.b)=>ADD->.c] ->
              .cell_definition(Factory)
                    -> Whatever ]
```

In this case, a "`Factory`" cell takes in a cell definition, feeds it to a primitive `NEW` cell which determines the underlying structure and generates a new cell according to the given definition. The new cell is then fed to some output facet. The `Test` cell here illustrates how the Factory is used to build a cell that computes a sum.

Example 3: dynamic instantiation with excretion and binding

We may now combine the preceding two concepts. The following example shows how a cell may accept another cell's definition, construct it, and bind it to one of its own exterior facets.

```
Colony [ ((.cell_definition -> NEW), .whichFacet) >> .east ]

Test [ (B[(a,.b)=>ADD->.c], FACET:".b") =>
            (.cell_definition, .whichFacet)(Colony:Y) ]
```

Facets themselves may be data and are identified symbolically. In the Colony cell, `>>` (the externalize-and-bond operator) is stimulated by the stimuli set <cell, facet> and pushes out a new cell and binds to .east of a Colony cell. In the Test cell we provided some cell B and the encoding of one of its facets to a new Colony named Y. The new cell's bond facet is specified by the input to `.whichFacet`, which in this case is `FACET:".b"`.

Comparisons

All binary comparisons have two input facets `(.left,.right)` and three output facets `(.T, .F, .output)`. The `.T` and `.F` facets are stimulated if the condition is true or false respectively. The `.output` facet returns 0 if false and 1 if true. The core comparisons are: `EQUALS`, `LESS_THAN`, `GREATER_THAN`

Non-Trivial Examples

Conway's Game of Life simulation

This is the cell of Conway's Game of Life, a cellular automaton in which:

> The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square *cells*, each of which is in one of two possible states, *live* or *dead*. Every cell interacts with its eight *neighbours*, which are the cells that are directly horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:
>
> 1. Any live cell with fewer than two live neighbours dies, as if by loneliness.
> 2. Any live cell with more than three live neighbours dies, as if by overcrowding.
> 3. Any live cell with two or three live neighbours lives, unchanged, to the next generation.
> 4. Any dead cell with exactly three live neighbours comes to life.
>
> -Wikipedia

```
Conway [(.UP, .DOWN, .LEFT, .RIGHT) -> /* Sum the 4 sides*/
     ADD -> (.left(LESS_THAN:LT), .left(GREATER_THAN:GT));
     /* feed the sum as the left side of comparison cells*/
          INT:2 -> .right(LT);     /* compare sum to 2 */
          INT:3 -> .left(GT); /* compare sum to 3 */
          (.output(LT), .output(GT))->(.UP,.DOWN,.LEFT,.RIGHT)]
          /* send the outputs of the comparators
            (0 or 1, false or true) to all 4 faces */
```

The Conway cell feeds its four sides into a sum, which is compared to 2 and 3. The outputs of the comparisons are outputted to the facets to indicate whether the cell dies or lives.

GCD algorithm

This cell is equivalent to the GCD algorithm represented by the equivalent C code:

```
int gcd(int a, int b) {
     if (a == 0) return b;
     else return gcd( b % a, a);
}
```

```
GCD[.a -> (.left(EQUALS:eq), .b);
      /* arg .a goes to comparator and to .b*/
      INT:0 -> .right(eq)(.T,.F) =>
      /* 0 is compared with .a and the results may stimulate
         one of two anonymous binder cells…*/
          ([.b -> .return],     /* return .b */
           [.a -> .R(%:mod)]); /* or send (b % a) back to .a */
    .b -> .L(mod).out -> .a ]
```