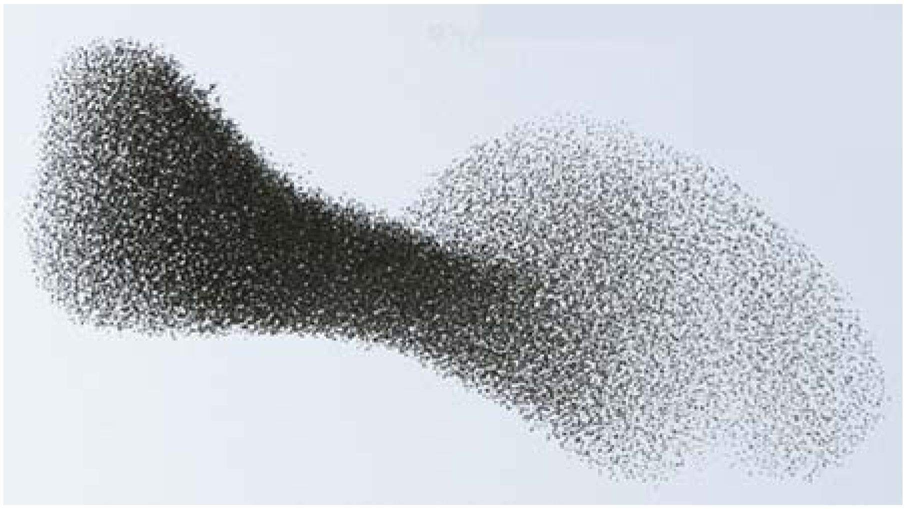
Swarm

Language Reference Manual [LRM]



A Cellular Programming Language

Team Epidemic

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.

Table of Contents

Introduction

Swarm's style is declarative, capturing the computation in a manner similar to that of functional languages. Its programs, rather than executing operations imperatively, define the juxtaposition of data and functionality explicitly. Swarm is a metalanguage whose operators perform high-level manipulations of the superstructure of a network of functional units. Swarm operators, when executed, link together these functional units, the *cells*, which provide conventional functionality. This introductory section will introduce four concepts: the Swarm model of computation, cells, facets, and cell bindings.

```
[STR:"Hello World!" -> STDOUT]:MAIN; /* Hello World! */
```

Model of Computation

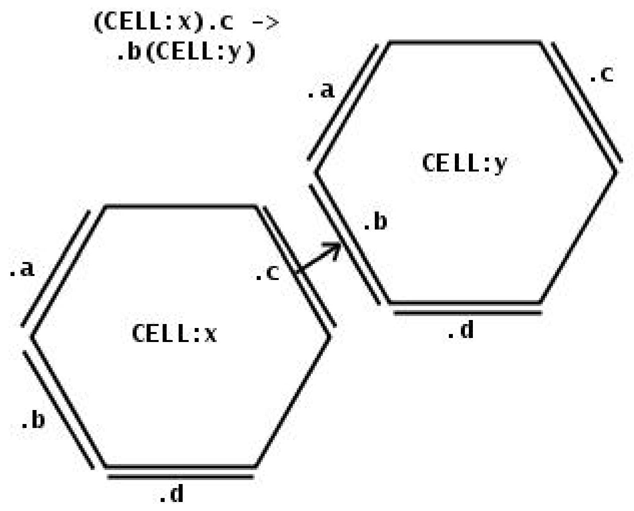


Figure 1: A hexagonal depiction of two cells and a bind.

Programs in Swarm are the structural definitions of cellular lattices. Figure 1A depicts a very simple lattice consisting of two cells and the snippet of code that describes it.

In Figure 1, assuming that the type named "CELL" is defined, the language construct "CELL:x" instantiates a new cell of that type and labels it with the symbol 'x'. One of its facets (named .c) is then bound to some facet (named .b) of another cell of the same type named 'y'.

When Swarm programs are run, a lattice of cells is constructed and some computation may begin when any of the cells receive stimuli (pieces of data emitted by another cell). All data is cellular in nature; the idea of "primitive types" is also considered as cellular. The initial stimulus may come from a MAIN cell that is automatically started, similar to main functions in C, or from the user via a GUI or standard input. Stimulation occurs when data is transferred from one cell to another via their facets.

The Cell

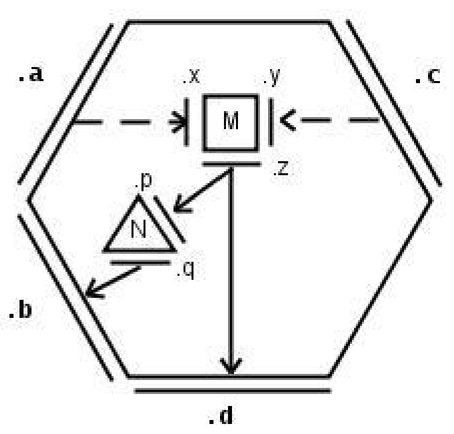


Figure 2: A Cell [and internal substructure]

The most important concept is that of the cell. It is the primary 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. To facilitate the construction of these graphs (which we shall refer to as *lattices*), cells' connectivity is declared in terms of links between facets (defined binding sites). Note that 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. After a cell performs a computation, its output values are propagated automatically through facets. Cells have a recursive relationship with each other in that cells may be composed to define composite cells. In such derived cells, the programmer specifies how sub-cells interface each other as well as how they interface with the membrane of the enclosing cell.

The Facet

Facets are points of control of the data flow. Binding between facets is not checked at compilation: there are no strict "binding types". However, a cell membrane's binding sites may filter the stimuli they receive, deciding whether to permit data inside the cell based on the data's value, properties, or type (see TYPE in the section Special Semantics). The cells are only stimulated to compute when the proper set of stimuli impinge on their binding sites. The stimulating data then "binds and waits", attaching to the facet until the facet is ready to internalize it. When the facet is ready, all values diffused into the cell and fed to the cell's interior lattice of sub-cells, which will perform a computation and possibly propagate values out its facets as output. More details on how a lattice executes and propagates the computation is explained in the section "Execution Flow".

Binding

A binding defines a flow of data from facet to facet. In general, arbitrarily many facets may bind to any one facet; also, facets may be used bidirectionally -- a facet can be specified to be used for emitting as well as receiving data. A programmer *positively defines* bindings between cells, such that in the case of undefined output flows, the values emitted are unused. That is, if some cell defines behavior for both input and output to one of its facets, the programmer can bind another cell to it unidirectionally; **the availability of an opposite direction does not force the programmer to utilize both. See the section on binding operators for implementation details.**

Lexical Conventions

Swarm has no keywords. Whitespaceiscompletelyignored!

Comments

The characters `/*` introduce a comment, which terminates with the characters `*/`. Comments do not nest, and they do not occur within string literals.

Identifiers

There are three types of identifiers.

1. A CELLTYPE, which serves as a label for the structural definition of a cell, is a sequence of one or more uppercase letters.

2. An INSTANCE is a sequence of one or more lowercase letters or a sequence of one or more digits, but not both. An INSTANCE labels a particular concrete instance of some cell.

3. A FACET is a '.' followed by a sequence of one or more lowercase characters. When succeeded by a CELLTYPE or INSTANCE in parens (e.g. .input(MYCELL), the FACET is considered to be contextualized to that cell. Likewise, the FACET may be preceded (e.g. MYCELL.output). Allowance of both techniques allows for chaining of bind operations. When alone, the FACET is contextualized to the nearest enclosing named cell definition.

Special Semantics

Swarm provides certain special built-in cells. They have a special-case semantic regarding the identifier to the right of the colon. They use these identifiers as qualifiers or as literals to complete the cell's behavior. For example, the STR cell type needs a string literal in order to be instantiated and to have a value to propagate.

Primitive data cells:

Primitive data type cells have one single input facet, (.to), and one output facet, (.out). and can be instantiated using the : operator. The input facet, (.to), converts an input passed to that facet into that data type. For example, [STR:"42.0"->FLT->SOUT] will convert the STR "42" into the FLT 42.0, then print it out. The output facet, (.out), outputs this cell.

INT

INT is the equivalent of "int" in C.
Example instantiation: `INT:”-5”`

FLT

FLT is the equivalent of "float" in C.
Example instantiation: `FLT:”4.57”`

BOOL

BOOL is the equivalent of an unsigned single-bit integer in C. INT and FLT converted into a BOOL will be 1 unless they equal 0. Unless it is an empty string, any STR converted into a BOOL will be 1.
Example instantiation: `BOOL:”1”`

STR

STR is the equivalent of "String" in Java. A BOOL converted to a STR will appear as "0" or "1".
Example instantiation: `STR:"Hello Swarm!\n"`

Other Special Cells:

SIN

SIN parses the first token from STDIN as a specified type. If the input is not of the specified type, SIN will block until a valid input is given. SIN has one output facet, (.out), whose output is of the specified type. The format is SIN:<type>, where <type> is the lowercased name of the type of cell to read.
Example: `SIN:int` will output the first valid INT from STDIN.

```
>”10 apples and 10 bananas” [SIN:int will spit out 10]
```

TYPE

TYPE has a single input facet, (.in), and a single output facet, (.out). TYPE will check if a given input is of a specified cell type, and propagate the input on (.out) if it is. If not, TYPE will block and not output. The format is TYPE:<type>, where <type> is the lowercased name of the type of cell to check against.
Example: `INT:12->TYPE:str->SOUT` will block because INT is not of type STR.

MAIN

MAIN is a very special purpose cell which identifies to the compiler where the actual program instruction sequence lies. There must be exactly one MAIN cell in any program, and MAIN cells in included files will be ignored. Main does not have any input or output facets.
Example: `[INT:42->STR->SOUT]:MAIN` defines a program that prints the STR "42" to STDOUT.

Meaning of Identifiers

The CELLTYPE, or cell, is a functional unit that can be bound to other functional units by one or more Facets that belong to it. Its structure is encoded between square brackets.

The INSTANCE is an identifier that corresponds to a particular concrete instance of one type of cell. If it is addressed in multiple statements or in multiple bindings, the operators will act upon the same object. When a cell is not labeled in this manner, every use of its type in a binding relationship will assume a distinct and nameless new cell of that type.

The FACET is an identifier which symbolically references a particular named facet of either the contextualizing cell or of the enclosing cell. A facet is contextualized by a following cell identifier enclosed in parentheses or by a a preceding cell identifier optionally enclosed in parentheses. Groups of facets can be referenced and contextualized; in such cases, the facets in the group must be enclosed in parentheses and delimited by commas. The group is contextualized in the same manner as individual facets relative to a cell identifier.

Generally: prefixing enforces enclosing of the cell identifier in parentheses. Cells may be surrounded by any combination of individuals or groups.

Non-exhaustive Set of Examples of Valid forms:

```
.facet(CELL)  singular prefixed facet
CELL.facet    singular postfixed facet
(.faceta, .facetb)(CELL)(.facetc, .facetd)  postfixed and
prefixed groups of facets
```

The facets are notated in both prefix and postfix forms in order to facilitate "chains" of binding operations. See section on **Binding Operators** for an explanation of why prefix and postfix exist and how they are used.

Predefined Cells

Swarm includes several cells that are part of the core library. The descriptions are below:

Math cells:

ADD

ADD takes in n > 0 arguments and adds them. If any arguments are of type INT, it will output an INT. If either or both inputs are of type FLT, it will output a FLT. The individual input facets of ADD cannot be accessed individually, due to the nature of the cell.

```
<(*) ADD (.out)>
```
*: any number greater than 0 of INT or FLT inputs
out: if any input is of type FLT, out is of type FLT. Otherwise, it is of type INT

SUB

SUB takes in two arguments and subtracts the second from the first. If both arguments are of type INT, it will output an INT. If either input is of type FLT, it will output a FLT.

```
<(.a, .b) SUB (.out)>
```
a: an INT or FLT
b: an INT or FLT
out: if either a or b are of type FLT, out is of type FLT. Otherwise, it is of type INT

MULT

MULT takes in n arguments and multiplies them. MULT will always output type FLT. The individual input facets of ADD cannot be accessed individually, due to the nature of the cell.

```
<(*) MULT (.out)>
```
*: any number greater than 0 of INT or FLT inputs
out: an FLT

DIV

DIV takes in two arguments and divides the first by the second argument. MULT will always output type FLT.

```
<(.a, .b) DIV (.out)>
```
a: an INT or FLT
b: an INT or FLT
out: an FLT

POW

POW takes in two arguments and raises the first to the power of the second. Fractional and negative powers are accepted. POW will always output type FLT.

```
<(.a, .b) POW (.out)>
```
a: an INT or FLT
b: an INT or FLT
out: an FLT

MOD
MOD takes in two arguments and computes the modulus, where the first argument is the dividend and the second argument is the divisor. MOD will always output type INT.

```
<(.a, .b) MOD (.out)>
```
a: an INT
b: an INT
out: an INT

Comparison Cells:

CMP
CMP takes in two arguments and compares them. Both arguments must be of the same type for comparison to occur, and the output will be an INT. If the first argument is smaller than the second argument, the output will be -1. If both arguments are equal, the output will be 0. If the first argument is greater than the second argument, the output will be 1.

```
<(.a, .b) CMP (.out)>
```
a: an INT, FLT, BOOL, or STR, must be same type as b
b: an INT, FLT, BOOL, or STR, must be same type as a
out: an INT

NAND
NAND takes in two BOOL arguments, and returns the NAND of the arguments in a BOOL.

```
<(.a, .b) NAND (.out)>
```
a: a BOOL
b: a BOOL
out: a BOOL

Input / Output Cells:

SOUT
SOUT takes in an argument and prints it out onto standard output. If the argument is not

an STR it will be converted to an STR before printing.
```
<(.data) SOUT>
```

data: an INT, FLT, BOOL, or STR

FIN
FIN reads in a file specified by an STR argument and returns the data contained in that file as a STR.
```
<.filepath FIN .out>
```

.filepath: a STR

FOUT
FOUT writes an STR argument to a filepath specified in another STR argument.
```
<(.filepath, .data) FOUT>
```

.filepath: a STR
.data: a STR

INCLUDE

INCLUDE takes a single STR and imports the specified .swm file and any declarations within.

```
<(.filepath) INCLUDE>
```

.filepath: a STR

DATA:
TYPE

TYPE takes a single PRIMITIVE and creates a cell that blocks data that isn't of type PRIMITIVE. If the data fed into the TYPE cell is not of the required type, the cell will block and not output, otherwise it will pass the data onto the out facet.

```
<(.data) TYPE:PRIMITIVE (.out)>
```

.data: an INT, FLT, BOOL, or STR

PRIMITIVE: "str", "int", "flt", or "bool"
.out: same type as .data if it matches PRIMITIVE, else no output

Derived Cells

Derived cells arise from compositions of fundamental cells and other derived cells.

Derived cells are declared as the binding of other cells. The structural definition is enclosed in square brackets and, if the cell is named, followed by a cell type name in all capital letters. All uncontextualized facets in are assumed to be owned by the innermost enclosing named cell. There are two types of derived cells:

1. *Named cells* have a cell type identifier (the name) and named facets. The structural definition (enclosed in brackets) specifies how facets and any internal cells are linked.

2. *Anonymous cells* cannot have named facets. They may be declared within an enclosing cell definition. They also have a special semantics triggered by any input whatsoever; when stimulated, they temporarily alter the enclosing cell's binding according to the lattice defined within the anonymous cell's braces. The reconfiguration endures only for the current computation.

Conversions

Conversion between types of literals (or casting) is accomplished through use of the .to facet, which all built-in data types have.

Example: `INT:5->.to(STR)`
This example converts the int literal 5 to the string "5".

Declarations

Literals are defined through the special semantics of the INT, FLT, STR, BOOL cells. There is no concept of "Variable" in the C-sense. The analogue of the variable is the facet, which take in data and are specified at the outset. There are some implicit declarations using the `SIN:type` and `FIN:type` syntax

Statements

Cellular lattice definitions consist of a list of statements separated by semicolons, or optionally terminated by semicolons. Each statement is one chain of facets or cells bound to other facets or cells. The binds between blocks define the nature of connections. To resolve more complicated loops [cells which take in multiple inputs in different chains], multiple statements can refer to the same objects. For example:

```
.a -> CELL:b -> .c; .c -> b;
```

The two statements above refer to the same cell instance, labeled as b

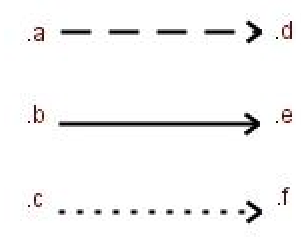
Binding Operators

The binding operators allow cells to be linked together through their facets. There are two types of binding, direct binding and cross binding.

=> [Direct Bind]

This binds groups of facets element-wise, such that the nth facet in one group binds the nth facet in another group. For example:

```
    (INT:3, INT:4) =>  (.a, .b)(HYPOTENUSE)
```



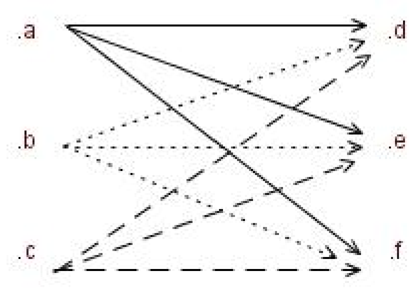
This binds the primitive cells for integers 3 and 4 to the .a and .b facets, respectively.

-> [Cross Bind]

This binds elements in groups of facets in a cross-product. In graph theoretical terms, it produces the complete bipartite graph $K_{n,m}$ where n and m are the number of facets in the first and second groups respectively. Binding two cells directly will perform the equivalent operation on their entire facet sets. For example:

```
  (.a,.b,.c) -> (.d, .e, .f)
```

produces:



-| [Kill]

This bind will halt the execution in this statement chain. The difference between using the kill bind and just stopping the chain is that a labeled cell will be eliminated. Hence:

```
  .a -> b;                              .a -> b -|
```

At the end of the first statement, b still resides [for example, another statement can use]. On the other hand, the kill ensures that b will be destroyed.

>> [Push]

This bind will, upon creation of a cell, push the cell out and bind it to the parent cell. The main usage is in anonymous and dynamically created cells:

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

At the end of this statement, the anonymous cell [of type R] is pushed out and bonded to the parent cell [.east of the parent cell is bound to the .west of the new cell]. Through this, we can create linked lists and other data structures.

Chaining

To permit succinct expression of multiple relationships, the binding operators may be combined into chains, in which the postfixed facets of a cell are bound to the prefixed facets of another cell. A list of several statements, each containing a chain, constructs a full cell definition and describes a cellular structure. An example of a chain:

```
.a(A).b -> .c(D).e -> (.f(G).h, .i(J).k) => L
```

Execution Flow

A program is defined as the contents of the MAIN cell. Programs execute as the propagation of data, which flows along the paths positively declared by the programmer. However, not all data will be propagated along all paths in sync. Therefore, a facet requires that all required inputs impinge upon it before it triggers a computation within the cell. For example, if a facet requires three integers, and a programmer had three cells of differing complexity feed it, the facet would wait until all three feeders had completely sent their values before initiating a calculation. The earliest values simply "bind and wait," to be internalized only after the last value arrives. Once the facet triggers the cell, the outputs of that computation will be released simultaneously (if it branches inside and flows to different output facets); outbound values will also have the "bind and wait" property. Discrepancies in cells' "speed" are spooled in this way. Conceptually, facets represent the idea of data dependency by naturally expressing the points at which various task threads need to wait upon each other before progressing.

Because cells will only compute when stimulated with valid inputs, program "flow" is controlled by whether cells propagate outputs or not. An example of this can be seen in the TYPE cell. Entire regions of a cell may not be activated. A cell's flow may also be reconfigured by the activation of anonymous cells that impose new bindings.

Scope and Linkage

All code is defined in `*.swm` files. A `*.swm` file consists of any number of cells depending on the programmer's choice. Each file can only contain one main cell, and main cells of included files are ignored. The syntax for including other files is:

```
STR:"lib.swm"->INCLUDE;
```

where this passes the list of cells in "`lib.swm`" to the include cell. Now all those cells can be referenced in the current file.

Grammar

The following grammar is derived from our .g files:

```
END: ';';
NEW: ':';
LPAREN: '(';
RPAREN: ')';
LBRACKET: '[';
RBRACKET: ']';
COMMA: ',';

CROSSBIND: "->";
DIRECTBIND: "=>";
KILL: "-|";
PUSH: ">>";
BINDTYPE: CROSSBIND | DIRECTBIND;

DIGIT: ('0'..'9');
UPPER: ('A'..'Z');
LOWER: ('a'..'z');
DIGIT_HEX : (DIGIT|'a'..'f'|'A'..'F') ;

OCTAL: ('0'..'3') (DIGIT (DIGIT)? )? | ('4'..'7') (DIGIT)?;
UNICODE : 'u' DIGIT_HEX DIGIT_HEX DIGIT_HEX DIGIT_HEX;

WS: ( ' ' | '\t' | "\r\n" | '\r' | '\n');
ESCAPE :   '\\' (('b'|'t'|'n'|'f'|'r'|'"' /* " */ |'\''|'\\')
       | OCTAL | UNICODE);
```

```
STRING :  '"'! ( ESCAPE | ~('\\'|'"') )* '"'! ;

COMMENT: "/*"
         ( options { generateAmbigWarnings=false; }
           : { LA(2)!='/' }? '*'
           | '\r' '\n' {newline();}
           | '\r' {newline();}
           | '\n' {newline();}
           | ~('*'|'\n'|'\r')
         )*
         "*/" {$setType(Token.SKIP);}
         ;

CELLTYPE: (UPPER) ((UPPER) | '_' )* ;
INSTANCE: (LOWER)+ | (DIGIT)+ ;
FACET: '.' (LOWER) ((LOWER) | '_' | (DIGIT))* ;

/* Plain old Facet */
facetunit  : FACET;

/* Facet with post - (identification | definition)
 * Added to resolve .a -> (.left(EQUALS:eq), .b) */
facetdef : facetunit (LPAREN! unit RPAREN! (facetunit)?)?;

/* Isolated or groups of facets */
facetblock : facetunit
           | LPAREN! facetdef (COMMA! facetdef)* RPAREN!;

/* Cell-level specifiers */
protected
unit : CELLTYPE (NEW! INSTANCE
         | NEW! STRING)
     | anon_def;

/* Statement-level blocks */
block: facetblock (LPAREN! unit RPAREN! (facetblock)?) | unit
     | LPAREN! unit ( (RPAREN! (facetblock)? )
                    | ((COMMA! unit)+ RPAREN!));

/* Statements */
statement: block (bindtype block)* ;

/* Anonymous Cells */
anon_def: LBRACK! statement (END! statement)* (END!)? RBRACK!

/* Cell Definitions */
definition: anon_def NEW! celltype END!
```