

Echelon

A Language for Human-Written Fleet Programs

Adam Megacz

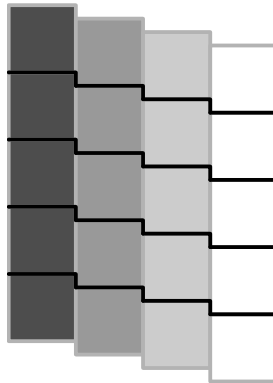
August 14, 2007

Abstract

Echelon is a high-level language which can be used to write efficient programs for Fleet. It is closely matched with Fleet's abilities, yet is expressive enough for humans to write programs in by hand.

1 An Echelon: The Basic Unit of Computation

The basic unit of computation in Echelon is the echelon (plural: echelons). It consists of a table-like structure of data having a fixed, finite width and either a fixed height or unbounded height:



We draw the items in a row in a staggered configuration in order to emphasize the fact that they are processed in strictly left-to-right order (think of the echelon as moving upwards). This encourages the programmer to arrange the columns of her echelons so that their left-to-right order matches the order in which they are processed.

We use the same shading color for each column to indicate that each column of data must be treated identically. As we will see later in this document, the Echelon compiler compiles an operation on a w -wide echelon into a queueing

instruction loop of w instructions. For this reason, w must be known at compile time and every w^{th} item must be handled by the same instruction.

By contrast, echelons may be of either fixed height known at compile time (“finite echelons”) or of unbounded height (“infinite echelons”). These correspond to finite and infinite requeue counts on instructions, as we will also see later.

2 Syntax

The syntax and semantics of Echelon attempts to stay close to being a subset of Java wherever possible. However, there are five primary differences:

- All variables are implicitly `final` – they may only be assigned once. This ensures predictable, high-performance compilation.
- Types are written *after* the variables they apply to, separated by a colon. For example,

```
myInt : int;

myFunction(a:int, b:char) : int {
    ...
}
```

- Thanks to the previous item, function types may be written *explicitly*, using the `->` operator to separate the argument types from the return type. For example,

```
myFunction : int, char -> int;
```

- Functions may have multiple return values:

```
myOtherFunction : int, char -> char, int;
```

- A new type is introduced into the language, the type of echelons.

The type of finite echelons having width w and height h is written $[w/h]$. The type of infinite echelons having width w is written $[w/*]$.

Either w or h may be omitted if it is 1, so $[2/]$ is a two-wide, one-high echelon, $[/]$ is a one-wide, one-tall echelon (“unit echelon”), and $[/*]$ is a one-wide, infinitely tall echelon. As a special case, the programmer is allowed to write $[*]$ for the type of the one-wide, infinitely tall echelon.

3 \forall -quantification

If a single-letter, lower-case alphabetical character appears where a width or height is expected, this is an *implicitly \forall -quantified variable*. Although this may sound complicated, it is actually quite simple:

```
append : [a/b], [a/c] -> [a/(b+c)]
```

This type could be read as “the two arguments to append must have equal widths (a); the result will share that width and have a height equal to the sum (b+c) of the heights of the arguments.”

When the + operator appears in types, it might be applied to the infinite value *. We therefore declare that, for finite numbers n ,

- $n + * = *$
- $* + n$ is undefined
- $* + *$ is undefined

Note that this means that + is no longer strictly commutative. In the append example above, the second argument may be infinitely long, but the first must not be. This restriction may be eased in the future.

The operators max() and min() may also appear in numerical positions. Finally, arbitrary int arguments may be bound to \forall -quantified variables; for example, a function which takes an int and returns a one-wide echelons whose height is equal to the argument would have this type:

```
oneWideForHeight : h:int -> [/h];
```

Note that this is a *dependent type* [McK06]. We restrict the domain of dependent types to echelons whose sizes are equations in Presburger Arithmetic [Pre29] in order to ensure decidability of the type system.

4 Representing Echelons in Fleet

A [w/h] echelon is represented in fleet as a sequence of w*h data items moving between valves where each valve is executing a requeueing loop of w instructions with a requeue count of h. When h=*, these instructions requeue forever, and other means are used to signal the end of a sequence (using kill’s and extra tokens).

5 Basic Operators on Echelons

Here are the types of some primitive functions on echelons¹. Note how easy it is to determine what a function does simply by thinking about the shapes of the echelons in its type! For ease of reading, the types below do not use any abbreviations.

<code>unit</code>	:	<code>int</code>	->	<code>[1/1]</code> ;
<code>ununit</code>	:	<code>[1/1]</code>	->	<code>int</code> ;
<code>transpose</code>	:	<code>[a/b]</code>	->	<code>[b/a]</code> ;
<code>zip</code>	:	<code>[a/b], [c/d]</code>	->	<code>[(a+c)/max(b,d)]</code> ;
<code>unzip</code>	:	<code>a:int, [(a+b)/c]</code>	->	<code>[a/c], [b/c]</code> ;
<code>append</code>	:	<code>[a/b], [a/c]</code>	->	<code>[a/(b+c)]</code> ;
<code>unappend</code>	:	<code>b:int, [a/(b+c)]</code>	->	<code>[a/b], [a/c]</code> ;
<code>flatten</code>	:	<code>[a/b]</code>	->	<code>[(a*b)/1]</code>
<code>unflatten</code>	:	<code>a:int, [(a*b)/1]</code>	->	<code>[a/b]</code>
<code>loop</code>	:	<code>[a/b]</code>	->	<code>[a/*]</code> ;
<code>repeat</code>	:	<code>a:int, [b/c]</code>	->	<code>[b/a*c]</code> ;
<code>memRead</code>	:	<code>int[]</code>	->	<code>[1/*]</code>
<code>memWrite</code>	:	<code>[1/a], int[]</code>	->	<code>void</code>
<code>map</code>	:	<code>(int -> int)</code>	,	<code>[1/a]</code> -> <code>[1/a]</code>
<code>map2</code>	:	<code>(int,int -> int,int)</code>	,	<code>[2/a]</code> -> <code>[2/a]</code>

Dear reader, it's probably a good idea to stop, take a break, come back and carefully work through each of the previous types, making sure you understand at least the shape of the echelons involved. This should take about a half an hour, but it is absolutely essential to a proper understanding of the language. Don't skip this step!

A common design pattern is to unzip an echelon, map different functions onto its constituent columns, and then re-zip the columns back together.

Note that the two functions `memRead` and `memWrite` can be used to move data between `int[]`'s and echelons. At the moment, this is the primary mechanism for integrating Java programs and Echelon programs – the Java program will put data into an `int[]`, invoke an Echelon program which `memRead`'s it into the Fleet core, manipulates it, `memWrite`'s it out to memory, and then returns control to the Java program. More elegant mechanisms are planned for the (far) future.

¹FIXME: multiplication is not Presburger decidable!

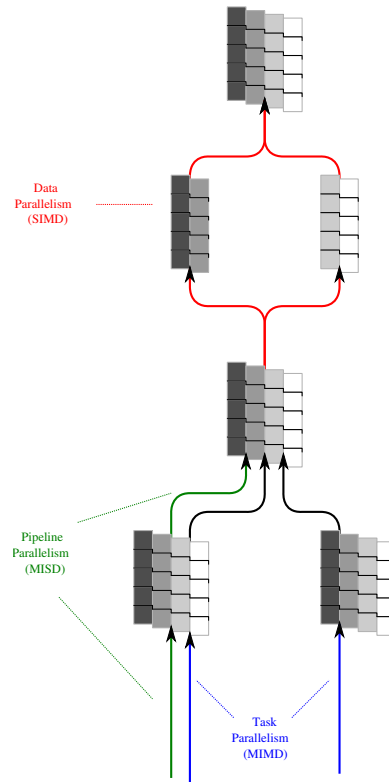
6 Exploiting Parallelism

Echelon is capable of exploiting all three major forms of parallelism: task parallelism (MIMD), pipeline parallelism (MISD), and data parallelism (SIMD).

Note that SIMD is typically used to describe flow graphs with a branch followed immediately by independent operators and then a join in which the “flow rate” on both sides of the branch are in a fixed relationship.

By contrast, MIMD applies to parallelism in its most general form, usually as stated explicitly by the programmer.

Pipeline parallelism appears many places, one of which being the ability to map one function onto an echelon, and then map another function onto the result. The two functions may be evaluated in pipeline-parallel (MISD) fashion.



7 Integration with Java

As described so far, echelons are composed strictly of primitive machine-words (ints); a row of an echelon cannot be treated as anything other than a collection of these. Clearly it would be desirable for Echelon to offer some greater level of abstraction; ideally echelons should be object-oriented, offering encapsulation, inheritance, and polymorphism for rows. Ideally, such an inheritance mechanism would be fully integrated with the Java class hierarchy, making Echelon a fully integrated extension of the Java language.

Unfortunately, as it currently stands, the Java class hierarchy is incapable of expressing this sort of structure. This is because Java objects are strictly *pass-by-reference*, with no support for “value objects”. An example helps here:

```
class Nancy {
    int i;
    char c;
}

class Fred {
    Nancy n1;
    Nancy n2;
}
```

In the Java language, the class Fred will be implemented as a heap object with pointers to two other distinct heap objects of type Nancy. By contrast, a C++ programmer could choose this implementation (by changing Nancy to Nancy*), or they could choose to have the Nancys *inlined* so that Fred is compiled as if it had been declared as:

```
class Fred {
    int nancy1_i;
    char nancy1_c;
    int nancy2_i;
    char nancy2_c;
}
```

The ability to “inline” classes in this fashion is essential for object-orientation in Echelon, because the primitive operators on echelons do not perform pointer-chasing – nor do we want them to, because this would make it difficult for the programmer to reason about the performance of code.

Fortunately, this deficiency in Java has been identified already by several researchers, and a number of solutions have been proposed. Of these solutions, by far the most beautiful and well-developed is the work of David Bacon on *Kava* [Bac01]²

²as an example of the level of elegance Kava offers, it reformulates the Java `int` as nothing more than a 32-element array of `bits`, where `bit` is an enum having two values. The ability to apply object-oriented principles is extended even to pre-existing Java primitive types!

Kava introduces a second branch of the class hierarchy, parallel to `Object`. This second branch is rooted at type `Value`, which is the superclass of all classes whose fields do not contain references. An ideal integration of Echelon with Java would start by adopting the Kava extensions, and then proceed to permit echelons whose rows may be of any subclass of `Value`; for example,

```
class Point3D extends Value {
    int x;
    int y;
    int z;
}

class Triangle extends Value {
    Point3D p1;
    Point3D p2;
    Point3D p3;
}

myEchelonOfTriangles : [Triangle/*];
```

Note, however, that this level of integration is an absolutely massive task, and involves a great deal of non-research implementation work. An initial prototype implementation of Echelon as a separate language is probably the best way to begin, with Kava integration as the end goal.

8 Comparison to Vector Languages

The main difference between Echelon and vector languages is the fact that Echelon's rows are *heterogeneous* – no assumption is made that the cells within a row are to be treated the same way, or even similarly.

The `unzip` and `zip` operators in Echelon are typically compiled into the *scatter* and *gather* idioms in Fleet assembly language (or Admiral [AK??]); support for these idioms at the microcode level makes it possible for Echelon to offer heterogeneous vector support at no additional cost.

9 Comparison to “Dataflow” languages

In theory, Echelon programs have data, and the data does in fact flow. In this sense, it might be fair to call Echelon a “dataflow language”, although under such a definition it would also be fair to apply the term to a host of other languages which are not generally considered “dataflow” languages (Haskell, for example).

In practice, the term “dataflow” is usually applied to two types of languages:

- **Static Dataflow languages.**

Static dataflow languages first appeared in Bert Sutherland’s thesis [Sut66], but were not fully explored and applied until the work of Jack Dennis [Den74]. These languages have a static, finite graph through which data items flow.

By contrast, in Echelon the programmer does not explicitly create the graph – it is inferred from declarative operators applied to echelons. Moreover, the Echelon analogue of a “flow graph” would change extremely often, even as often as once per datum processed, making the term “static dataflow” inappropriate.

- **Tagged-Token Dataflow languages.**

To date, all known non-static “dataflow” languages have used *tagged tokens* [Tra86]. This requires an *associative tag store*, which must be implemented in hardware as some sort of random-access device (such as an SRAM). A fundamental consequence of Logical Effort [SSH99] is that random-access devices have much higher logical effort than sequential-access devices, and therefore poorer performance.

Unlike tagged-token languages, Echelon encodes all “tagging” information into

1. The order of data items within a sequence
2. The order in which sequences arrive at and leave ships³.

This encoding is sound, because Fleet ensures that data items sent along a given path remain in sequence. This eliminates the need for associative tag stores, which were both the defining feature of and greatest flaw [CSEv93] in all previous non-static “dataflow” languages.

³when the last item of an echelon leaves a ship, it will typically cause the valve to move on to a new instruction loop (either because a requeueing count expired or due to a kill); this roughly corresponds to a restructuring of the graph in a dataflow language

10 Unfinished Ideas

Traditional architectures perform very poorly on pointer-chasing code (for example, traversing linked lists). It should be very easy to write a piece of code that, given the head of a linked list, will return an echelon representing the contents of the list – the memory latency penalty of the pointer chasing becomes parallel to all other processing. This packages a very powerful improvement in a way that is accessible to programmers.

There are actually two types of compile-time-unbounded (ie “infinite”) echelons: runtime-count-bounded (like Java arrays) and testable-terminator-bounded (like C strings).

In light of this programming model, it might be interesting to decouple manipulation of the valve’s count register from execution of instructions. For example, one type of instruction might load the count register with a value and then execute the following instruction *ignoring that instruction’s count*. When the count drops to zero, both instructions would be retired. This could also be used to remove the count field from the “instruction bit budget”.

Moreover, the mechanism in the previous paragraph can be used to implement a “finite repeating, infinite requeueing” instruction: both the “set count” and the “execute” instruction have a bit set that makes them requeue forever.

References

- [Bac01] David F. Bacon. Kava: a java dialect with a uniform object model for lightweight classes. In *Java Grande*, pages 68–77, 2001.
- [CSEv93] David E. Culler, Klaus Erik Schauer, and Thorsten Eicken von. Two Fundamental Limits on Dataflow Multiprocessing. In *Proceedings of the IFIP Working Group (Concurrent Systems) Conference on Architectures and Compilation techniques for fine and medium grain parallelism*. Elsevier, Jan 93.
- [Den74] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 362–376, London, UK, 1974. Springer-Verlag.
- [McK06] James McKinna. Why dependent types matter. *SIGPLAN Not.*, 41(1):1–1, 2006.
- [Pre29] M. Presburger. Ueber die vollstaendigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt. *Comptes Rendus du I congrÔs de MathÔmaticiens des Pays Slaves, Warsaw, Poland*, pages 92–101, 1929.

- [SSH99] Ivan Sutherland, Bob Sproull, and David Harris. *Logical effort: designing fast CMOS circuits*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.
- [Sut66] William R. Sutherland. *The On-Line Graphical Specification of Computer Procedures, Ph.D. Dissertation*. PhD thesis, MIT, 1966.
- [Tra86] K. R. Traub. A COMPILER FOR THE MIT TAGGED-TOKEN DATAFLOW ARCHITECTURE. Technical Report MIT/LCS/TR-370, 1986.