Euler: an applied lcc language for graph rewriting

Vijay Saraswat
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights
NY 10598
vijay@saraswat.org

Wed Mar 31 08:44:07 2004: Version 0.1 (vj): Basic note.

Abstract

We present a simple applied linear concurrent constraint programming (lcc) language, Euler, intended primarily for graph rewriting applications, as in formal molecular biology. The language permits set-forming operations in its term language, and the checking of (in-)equality constraints, but does not allow the imposition of equality constraints. Configurations may be understood as (hyper-)graphs, with atomic formulas representing nodes and logical variables representing (hyper-)edges, as usual. Rules may match and replace arbitrary (finite) subgraphs, while adding new nodes and edges.

The operational semantics of the language is specified by the general lcc framework, specialized to permit reasoning about additional operations.

We show that it can directly represent symbolic signaling pathways in molecular biology, of the kind studied by Danos and Laneve.

1 Euler

We introduce a simple applied lcc language [SL92,Tse92,FRS01], Euler, intended primarily for graph rewriting applications, as in formal molecular biology [DL04].

In such systems, the state of the computation is described by some kind of graph. (For simplicity, we consider simple directed graphs with binary edges.) Computation progresses via the application of graph rewrite rules, which are pairs (L, R) where L specifies a pattern that can be used to match a portion of the graph and delete it and R specifies the replacement graph. R may introduce new nodes, and new edges.

The abstract syntax of the language is given by the following grammar.

(Agent)	A	::=	$a(t_1, \ldots, t_n)$	(Atom)
			$A \otimes A$	(Parallel composition)
			$\exists x(A)$	(Local variable)
			R	(Rule)
			$!\, R$	(Replicated Rule)
(Rule)	R	::=	$(c \multimap A)$	(Implication)
			$\forall x(R)$	(Rule)(Quantification)
(Condition)	c	::=	$a(t_1, \ldots, t_n)$	(Atom)
			$t = t$	(Data decomposition)
			$c \otimes c$	(Parallel composition)
			$(\exists x)c$	(Local variable)
(Term)	t	::=	X	(Variable)
			$f(t_1, \ldots, t_n)$	(Uninterpreted functional terms)
			$\{\}$	(Empty set)
			$t : t$	(Set formation)

We permit usual first-order terms (as in Prolog), but reserve the binary infix term ":" to name a built-in set forming function. We shall write $\{t_1, \ldots, t_n\}$ for the term $t_1 : t_2 \ldots : t_n : \{\}$ and $\{t_1, \ldots, t_n \mid X\}$ for the term $t_1 : t_2 \ldots : t_n : X$.

We shall impose the restriction that if a term $t_1 : \ldots : t_k : t$ occurs in the body of a rule and t is a variable, then the variable is universally quantified in the rule and has a non-vacuous occurrence in the head of the rule. (Our intention is that only well-formed sets – that is, sets of the form $t_1 : \ldots t_k : \{\}$ – occur in a configuration.)

The first kind of agent is an atomic formula. Intuitively, atomic formulas (atoms) represent data-structures available in the "store". Different predicate symbols may be used to record the different state of the data structure. The second kind allows multiple agents to be run in parallel. The third kind allows ($\exists$-)abstraction over variables that are to be considered scoped to the given agent. The last allows us to specify a (possibly replicated) *reactive agent*. Such an agent can test whether its environment (the assembly of other agents running in parallel) satisfies a certain condition, and if so, can reduce to another agent.

As usual, $\exists$ and $\forall$ are binding operators. We identify formulas upto α-renaming.

Testing is performed by deductions in MLL, multiplicative linear logic. A test can ask whether the environment contains atoms that are connected to each other in specific ways or whether a data item matches a particular pattern.

1.1 Operational Semantics

The operational semantics is standard for lcc languages, with the built-in function symbols corresponding to set operations and the predicates corresponding to (in-)equality interpreted appropriately.

In more detail, a *configuration* is a multiset of agents (representing the "left hand side" of a sequent in MLL).

The transition relation on configurations is given by:

$$\Gamma, A \otimes B \longrightarrow \Gamma, A, B \tag{1}$$

$$\Gamma, (\exists X)A \longrightarrow \Gamma, A \quad (X \text{ not free in } A) \tag{2}$$

$$\frac{\Delta \vdash c[t_1, \ldots, t_n/X_1, \ldots, X_n]}{\Gamma, \Delta, \forall X_1, \ldots, X_n.c \multimap A \longrightarrow \Gamma, A[t_1, \ldots t_n/X_1, \ldots, X_n]} \tag{3}$$

$$\frac{\Gamma, R \longrightarrow \Gamma'}{\Gamma, !\, R \longrightarrow \Gamma', !\, R} \tag{4}$$

Above, $\vdash$ is the entailment relation for MLL, augmented with the following structural equivalence on terms and inference rules (for A atomic formulas):

$$t \equiv t \tag{5}$$

$$t_1 \equiv t_2, t_2 \equiv t_3 \Rightarrow t_1 \equiv t_3 \tag{6}$$

$$t_1 \equiv t'_1, \ldots, t_n \equiv t'_n \Leftrightarrow f(t_1, \ldots, t_n) \equiv f(t'_1, \ldots, t'_n) \tag{7}$$

$$t_1 \equiv t_2 \Rightarrow t_2 \equiv t_1 \tag{8}$$

$$t \equiv t', t_1 \equiv t'_1, t_2 \equiv t'_2 \Rightarrow t_1 : t_2 : t \equiv t'_1 : t'_2 : t' \tag{9}$$

$$\frac{t \equiv t'}{\begin{array}{l} a(t_1, \ldots, t_n)[t/X] \vdash a(t_1, \ldots, t_n)[t'/X] \\ \vdash t = t' \end{array}} \tag{10}$$

$$\vdash t \neq t' \quad (\not\vdash t = t') \tag{11}$$

As is standard in CCP, two agents are considered operationally equivalent if their sets of final stores are equivalent.[1]

Note that in the terminology of the π-calculus, we are permitting the "mismatch" operation. Euler does not permit any equality constraints to be added to the store. Hence it is permissible to respond to an $X \neq Y$ query successfully if X and Y are not structurally equivalent terms. (Note that two distinct variables are never structurally equivalent.)

2 Programming in Euler

A graph can be represented (up to isomorphism) as a collection of atomic formulas. For instance, a graph X consisting of three nodes, a, b and c, with edges $(a, b), (a, c)$ is represented by:

$$\exists A \exists B \exists c (e(A, B, X) \otimes e(A, C, X))$$

Transformations on graphs are represented by reactions.

Here is the representation of a rule (R_1) that creates a graph Y that is symmetric to X:

$$s(X, Y) \multimap\, !\, \forall A, B (e(A, B, X) \multimap e(B, A, Y))$$

The rule is triggered on the production of an atom $s(X, Y)$ in the configuration. The atom is consumed by the rule. Simulatenously, every atom $e(A, B, X)$ may be consumed and replaced by $e(B, A, Y)$. Computation terminates when all edges are so transformed (so that there is no edge left for X).

We make a distinction between the data-structure that is consumed and the data-structure that is produced in order to ensure that the program terminates in a finite number of steps. If the new edge is added to the same graph (X), the rule will trigger, thus causing an infinite loop.

If it is desired to keep the original graph, one may write instead the rule R_2:

$$s_0(X, Y, Z) \multimap\, !\, \forall A, B (e(A, B, X) \multimap e(A, B, Z) \otimes e(B, A, Y))$$

This copies the original graph from X to Z while producing the symmetric graph in Y.

[1] It should be possible to define a notion of barbed bismulation for the language. The connection between this operational congruence and that generated by the equivalence given here should be investigated for lcc languages in general.

One may specify a transformation R_3 that deletes edges while adding new ones:

$$t(X,Y) \multimap !\,\forall A, C(\exists B(e(A,B,X) \otimes e(B,C,X)) \multimap e(A,C,Y))$$

Consider the configuration:

$$e(A,B,X), e(B,C,X), e(C,A,X), t(X,X), R_3$$

It may reduce in one step to:

$$e(A,B,X), e(B,C,X), e(C,A,X), !\,\forall A, C(\exists B(e(A,B,X) \otimes e(B,C,X)) \multimap e(A,C,X))$$

Subsequently it may reduce to:

$$e(A,C,X), e(C,A,X), !\,\forall A, C(\exists B(e(A,B,X) \otimes e(B,C,X)) \multimap e(A,C,Y))$$

and then to

$$e(A,A,X), !\,\forall A, C(\exists B(e(A,B,X) \otimes e(B,C,X)) \multimap e(A,C,Y))$$

This is a terminal configuration.

2.1 Using sets

While the above representation of graphs is simple and convenient, it is sometimes necessary to have an explicit representation of the set of all edges emanating from a vertex.

2.2 Some useful idioms

First we develop some useful idioms in Euler. Below, we use the syntactic abbreviation $(\forall)(A)$ to indicate the universal closure of A with respect to all its free variables.

We may determine whether an item does not occur in a set as follows.

$$\begin{array}{l} !\,(\forall)(d(X,\{\},R,T,F) \multimap R:T) \otimes \\ !\,(\forall)(d(X,Y:Z,R,T,F) \otimes X \neq Y \multimap d(X,Z,R,T,F)) \otimes \\ !\,(\forall)(d(X,X:Z,R,T,F) \multimap R:F)) \end{array} \tag{12}$$

This program is invoked by adding the atom $d(X,S,R,T,F)$ to the store. On completion of the computation, either $R:T$[2] will be added to the store, or $R:F$. Thus the typical idiom for using this would be:

$$\begin{array}{ll} \exists R,T,F & (d(X,S,R,T,F) \\ & \otimes(R:T \multimap \ldots yes\ldots) \\ & \otimes(R:F \multimap \ldots no\ldots)) \end{array} \tag{13}$$

In what follows, we assume that the set of built-in predicates is extended with $\notin$ predicate: $X \notin S$ is true if S is a set and X is a term that does not occur in S, according to the definition above.

Next, we may iterate on all the items in a set as follows. Let A be an agent whose sole free variable is X. We define the agent $\forall X:t\ A$ to be the agent $(i(t) \otimes R)$ where R is:

$$\begin{array}{ll} !\,i(\{\}) \multimap 1 \otimes & \\ !\,(\forall)\ (i(T_1:T_2) \multimap \exists U & (b(U,T_1:T_2) \otimes \\ & b(U,\{\}) \multimap 1 \otimes \\ & \forall X,Y\ (b(U,X:Y) \multimap A \otimes b(U,Y)))) \end{array} \tag{14}$$

[2]Here we are using : /2 as a predicate symbol rather than a set forming operation.

Some idioms. We may swap edges as follows:

$$\forall E_1, E_2(v(X, Y : E_1) \otimes v(Y, E_2) \multimap v(X, E_1) \otimes v(Y, X : E_2)) \tag{15}$$

We may delete an edge as in the following example:

$$\forall E_1, E_2(v(X, Y : E_1) \otimes v(Y, Z : E_2) \multimap v(X, Z : E_1) \otimes v(Y, Z : E_2)) \tag{16}$$

Transitive closure. Now we may program transitive closure.

$$\begin{array}{rl} t(X) \multimap !\,(\forall) & (v(A, B_1 : E_1, X) \otimes v(B_1, B_2 : E_2, X) \otimes B_2 \notin B_1 : E_1 \\ & \multimap (v(A, B_1 : B_2 : E_1, X) \otimes v(B_1, B_2 : E_2, X)) \end{array}$$

By introducing an explicit check in the precondition of the rule we ensure that the program terminates after all required edges have been added.

2.3 Modeling protein signal cascades

We now have enough background to model signaling pathways as in [DL04].

A protein may have many receptor sites. We shall represent a protein as an atom with a single argument standing for the set of its receptor sites. This is convenient because a particular reaction names only a few receptor sites and by using set notation we will not have to enumerate the sites which are not involved in the reaction. We shall introduce a binary infix function symbol "." and write $a.v$ to denote that attribute a has value v. Finally we shall use the shorthand $s\{t_1, \ldots, t_n\}$ instead of $s(\{t_1, \ldots, t_n\})$.

A site s may be *free* or *bound*. A free site may be *visible* or *hidden*. If a site s is visible we shall represent it with the term s; if it is hidden, with the term $-s$ (where "$-$" is an uninterpreted unary function symbol) and if it is bound with the term $s.X$ where X is a variable which links this site to another site in another protein.

Consider the example of the signal cascade triggered by EGF [DL04, Sec 4.1]. There is a dimeric form EGF_2 of the growth factor EGF (written as the atom $s/1$ below) which binds two receptor EGFR proteins (written as $r/1$ below). The receptors cross-phosphorylate each other (through their second argument), allowing each to activate a second binding site (the third argument). This site binds to an adapter protein SHC (written as $a/2$ below), and activate it.

The signal-receptor interaction may be modeled through the following reactions:

$$\begin{array}{ll} (r_1) & !\,(\forall)(s(1 : S_1) \otimes s(1 : S_2) \multimap \exists X\ (s(1.X : S_1) \otimes s(1.X : S_2)) \\ (r_2) & !\,(\forall)(s(2 : S_1) \otimes r(1 : S_2) \multimap \exists X\ (s(2.X : S_1) \otimes r(1.X : S_2)) \end{array} \tag{17}$$

The RTK cascade is given by:

$$\begin{array}{ll} (r_3) & !\,(\forall)(s(1.X : 2.Y : S_1) \otimes s(1.X : 2.Z : S_2) \otimes r(1.Y : S_3) \otimes r(1.Z : -2 : S_4) \\ & \quad \multimap s(1.X : 2.Y : S_1) \otimes s(1.X : 2.Z : S_2) \otimes r(1.Y : S_3) \otimes r(1.Z : 2 : S_4)) \\ (r_4) & !\,(\forall)(r(2 : -3 : S) \multimap r(2 : 3 : S)) \\ (r_5) & !\,(\forall)(r(3 : S_1) \otimes a(1 : S_2) \multimap \exists Z\ (r(3.Z : S_1) \otimes a(1.Z : S_2))) \\ (r_6) & !\,(\forall)(r(2 : 3.X : S_1) \otimes a(1.X : -2 : S_2) \multimap r(2 : 3.X : S_1) \otimes a(1.X : 2 : S_2)) \end{array} \tag{18}$$

Now we can see this signal cascade in action. Computation is initiated on the presentation of two s proteins, two r proteins and one s proteins, all of whose sites are free:

$$s\{1, 2\} \otimes s\{1, 2\} \otimes r\{1, -2, -3\} \otimes r\{1, -2, -3\} \otimes a\{1, -2\} \tag{19}$$

The computation progresses as follows:

$$
\begin{array}{ll}
s\{1,2\}, s\{1,2\}, r\{1,-2,-3\}, r\{1,-2,-3\}, a\{1,-2\} & (20) \\
\longrightarrow^* s\{1.X,2\}, s\{1.X,2\}, r\{1,-2,-3\}, r\{1,-2,-3\}, a\{1,-2\} & (r_1) \\
\longrightarrow^* s\{1.X,2\}, s\{1.X,2.Y\}, r\{1.Y,-2,-3\}, r\{1,-2,-3\}, a\{1,-2\} & (r_2) \\
\longrightarrow^* s\{1.X,2.Z\}, s\{1.X,2.Y\}, r\{1.Y,-2,-3\}, r\{1.Z,-2,-3\}, a\{1,-2\} & (r_2) \\
\longrightarrow^* s\{1.X,2.Z\}, s\{1.X,2.Y\}, r\{1.Y,2,-3\}, r\{1.Z,-2,-3\}, a\{1,-2\} & (r_3) \\
\longrightarrow^* s\{1.X,2.Z\}, s\{1.X,2.Y\}, r\{1.Y,2,3\}, r\{1.Z,-2,-3\}, a\{1,-2\} & (r_4) \\
\longrightarrow^* s\{1.X,2.Z\}, s\{1.X,2.Y\}, r\{1.Y,2,3.U\}, r\{1.Z,-2,-3\}, a\{1.U,-2\} & (r_5) \\
\longrightarrow^* s\{1.X,2.Z\}, s\{1.X,2.Y\}, r\{1.Y,2,3.U\}, r\{1.Z,-2,-3\}, a\{1.U,2\} & (r_6)
\end{array}
$$

More complicated interactions may be represented in a similar fashion.

References

[DL04] V. Danos and C. Laneve. Formal Molecular Biology. *TCS*, 2004. To appear.

[FRS01] Francois Fages, Paul Ruet, and Sylvain Soliman. Linear concurrent constraint programming: operational and phase semantics. *Journal of Information and Computation*, 165(1):14–41, February 2001.

[SL92] Vijay Saraswat and Patrick Lincoln. Higher-order Linear Concurrent Constraint Programming. Technical report, Xerox PARC, 1992.

[Tse92] Clifford Tse. Linear Janus: A concurrent programming language. Technical report, Xerox PARC, Palo Alto, CA, 1992. MIT AI Lab MS Thesis.