A Parallel Programming Style and Its Algebra of Programs

Chris Hankin[1], Daniel Le Métayer[2], David Sands[3]

[1] Department of Computing, Imperial College
180 Queen's Gate, London SW7 2BZ, UK. (e-mail: `clh@doc.ic.ac.uk`)
[2] IRISA, Campus Universitaire de Beaulieu
35042-Rennes Cédex, FRANCE. (email:`lemetayer@irisa.fr`)
[3] Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø, DENMARK. (e-mail: `dave@diku.dk`)

Abstract. We present a set of primitive program schemes, which together with just two basic combining forms provide a suprisingly expressive parallel programming language. The primitive program schemes (called *tropes*) take the form of parameterised conditional rewrite rules, and the computational model is a variant of the *Gamma* style, in which computation proceeds by nondeterministic local rewriting of a global multiset.
We consider a number of examples which illustrate the use of tropes and we study the algebraic properties of the sequential and parallel combining forms. Using the examples we illustrate the application of these properties in the verification of some simple program transformations.

1 Introduction

In his Turing Award lecture, John Backus [Bac78] advocated the design of programming languages in terms of a fixed set of high-level constructors, or combinators, capturing common computation patterns. In this paper we follow Backus' programme in the context of parallel programming languages. This approach has two main benefits: it leads to more hierarchical and more structured programs; The set of combinators can be associated with a rich algebra of programs giving rise to useful program transformations. Backus exemplified his recommendations with the FP language. FP is a functional language based on a single data structure, the sequence. A set of combining forms is provided as the only means of constructing new programs from primitive functions. Examples of combining forms are:

$$\alpha f :< x_1, ..., x_n > = < f : x_1, ..., f : x_n > \qquad (f \circ g) : x = f(g : x)$$

Where : denotes application. The following is a typical law of the FP algebra: $\alpha(f \circ g) = (\alpha f) \circ (\alpha g)$.

[1] Partially funded by ESPRIT WGs 6345 (SemaGraph) and 6809 (Semantique).
[2] Partially funded by SERC Visiting Fellowship GRH 19330.
[3] Partially funded by ESPRIT WG 6809 (Semantique) and the Danish Natural Sciences Research Council

Most existing functional languages do not follow such a radical approach and allow the programmer to define his own combining forms as higher-order functions. Nevertheless the extensive use of higher-order functions is a main feature of the functional programming style because it favours readability, modularity and program reuse [Hug89].

Functional languages have also been claimed as attractive candidates for programming parallel machines. This belief is based on the fact that functional programs are free of side-effects. However functional languages encourage the use of recursion both for defining data structures and programs. In fact recursion is the only means of defining new data structures in functional languages. The problem with recursion is that it introduces an inherent sequentiality in the way data structures can be processed. For example if lists are defined as $\mathit{list}\ \alpha = \mathit{nil} \mid \mathit{cons}\ \alpha\ (\mathit{list}\ \alpha)$ then any list manipulating program must access the elements of the list sequentially. It may be the case that this sequentiality is not relevant to the logic of the program; so even functional languages can impose the introduction of unnecessary sequentiality in a program. This drawback is also identified in [Maa92] where a range of "parallel" datatypes are introduced.

To avoid this flaw, we propose a programming style based on multisets because the multiset (or bag) is the least constraining data structure possible. Starting with a recursively built data structure on a base type α (for example $\mathit{list}\ \alpha$) and removing all the structure, produces a multiset of elements of type α (not a set because the same element can occur several times in the structure). If the structure of the data is relevant to the logic of the program then it can be maintained in the multiset through a systematic encoding. For example a list $(x_1, ..., x_n)$ can be represented as a multiset of pairs (i, x_i). The important point is that this does not introduce any hierarchy between the values in the structure; all the values are still available and accessible at the same level.

For similar reasons, we do not allow recursion in programs since this encourages sequential hierarchy between computations. We focus in this paper on a collection of parameterised rewriting rules which form the counterpart of Backus's combining forms for multiset rewriting. These rules arise from previous work on the Gamma formalism [BM93]. We show the usefulness of these rules in terms of parallel program construction and we study their algebra of programs.

Section 2 is an informal introduction to the program schemes presented in this paper. Their semantics are defined formally in Section 3. Section 4 is a study of the confluence and termination of these schemes, describes the associated algebra of programs, and presents some examples of their use. The conclusion is a discussion of related works and avenues for further research.

2 Tropes: an Informal Introduction

We use a notation of the rewriting systems literature to denote multiset rewriting. The rule:

$$x_1, ..., x_n \rightarrow A(x_1, ..., x_n) \Leftarrow R(x_1, ..., x_n)$$

can be interpreted informally as the replacement into the multiset of elements $x_1, ..., x_n$ satisfying the condition $R(x_1, ..., x_n)$ by the elements of $A(x_1, ..., x_n)$. Viewed as a single program the (in general nondeterministic) result is obtained when no more rewrites can take place. A and R are functions. For example the program: $x_1, x_2 \rightarrow x_1 \Leftarrow (x_1 = x_2)$ computes the underlying set of a multiset by replacing pairs of equal elements by a single one until there are no further duplicate elements.

We consider only the rewriting of a single multiset in this paper and programs are not strongly typed (elements of different types can coexist in the multiset). When $R(x_1, ..., x_n) = True$, we omit the condition and write:

$$x_1, ..., x_n \rightarrow A(x_1, ..., x_n)$$

Similarly if $A(x_1, ..., x_n) = \emptyset$ we write: $x_1, ..., x_n \rightarrow \Leftarrow R(x_1, ..., x_n)$

We use two operators for combining programs, namely sequential composition $P_1 \circ P_2$ and parallel combination $P_1 + P_2$. Their semantics are defined formally in the next section. Not surprisingly, the intuition behind $P_1 \circ P_2$ is that the result of P_2 is passed as an argument to P_1. On the other hand, the result of $P_1 + P_2$ is obtained by performing the rewrites of P_1 and P_2 in parallel. For example, the following program returns the number of positive values in the initial integer multiset: $(x_1, x_2 \rightarrow x_1 + x_2) \circ ((x \rightarrow 1 \Leftarrow x > 1) + (x \rightarrow \Leftarrow x < 0))$.

The interested reader can find more examples of the relevance of multiset rewriting for parallel programming in [BM93]. We focus in this paper on five parameterised rewrite rules called *tropes* for:

Transmuter, **R**educer, **OP**timiser, **E**xpander, and **S**elector.

The tropes are defined in terms of multiset rewrites in the following way:

$$\begin{array}{ll}
\mathcal{T}(C, f) & = x \rightarrow f(x) \Leftarrow C(x) \\
\mathcal{R}(C, f) & = x, y \rightarrow f(x, y) \Leftarrow C(x, y) \\
\mathcal{O}(<, f_1, f_2, S) & = x, y \rightarrow f_1(x, y), f_2(x, y) \Leftarrow (f_1(x, y), f_2(x, y)) < (x, y) \\
 & \qquad \qquad \quad \textit{and } S(x, y) \textit{ and } S(f_1(x, y), f_2(x, y)) \\
\mathcal{E}(C, f_1, f_2) & = x \rightarrow f_1(x), f_2(x) \Leftarrow C(x) \\
\mathcal{S}_{i,j}(C) & = x_1, ..., x_i \rightarrow x_j, ..., x_i \Leftarrow C(x_1, ..., x_i) (\textbf{where } 1 < j \leq i + 1)
\end{array}$$

Notice that the reducer and the selector strictly decrease the size of the multiset; the expander increases its size; the transmuter and the optimiser keep its size constant. We first provide the intuition behind each of these tropes, then we present examples of composition of tropes and suggest the algebraic laws that they should satisfy.

Transmuter: The transmuter applies the same operation to all the elements of the multiset until no element satisfies the condition C. For example the following program returns, for each initial triple $(n, m, 0)$, a triple whose third component records the number of times m is a multiple of n.

$$nt \;=\; \mathcal{T}(C, f) \ \textbf{where}\, C((n, m, k)) = \textit{multiple}(m, n)$$
$$\qquad\qquad\qquad\qquad f((n, m, k)) \;= (n, m/n, k + 1)$$

Note the standard use of pattern matching in the definitions of C and f.

Reducer: This trope reduces the size of the multiset by applying a function to pairs of elements satisfying a given condition. The counterpart of the traditional functional *reduce* operator can be obtained with an always true reaction condition. For instance: $add = \mathcal{R}(True, +)$ returns the sum of the elements of a multiset.

Expander: The expander is used to decompose the elements of a multiset into a collection of basic values. For example *ones* decomposes positive values n into n occurrences of 1s.

$$\begin{aligned} ones \ = \ \mathcal{E}(C, f_1, f_2) \ \textbf{where}\, & C(x) = x > 1 \\ & f_1(x) = x - 1 \quad f_2(x) = 1 \end{aligned}$$

A multiset version of *iota*, which in FP generates a sequence from 1 to n, can be defined as:

$$\begin{aligned} & iota(n) \ = \ (\mathcal{T}(C_1, f_1) \ + \ \mathcal{E}(C_2, f_2, f_3)) \ \{(1, n)\} \quad \textbf{where} \\ & \quad C_1((x, y)) = (x \ = \ y) \quad f_1((x, y)) = x \\ & \quad C_2((x, y)) = (x \ \neq \ y) \\ & \quad f_2((x, y)) \ = (x, y - 1) \quad f_3((x, y)) = y \end{aligned}$$

Selector: The selector acts as a filter, removing from the multiset elements satisfying a certain condition. For example: $max \ = \ \mathcal{S}_{2,2}(\leq), rem \ = \ \mathcal{S}_{2,2}(multiple)$. So max returns the maximum element of a multiset and rem removes any element that is the multiple of another element ($multiple(x, y)$ is true if x is a multiple of y). The multiset of prime numbers smaller than n can be computed as:

$$primes \ = \ rem \ \circ \mathcal{S}_{1,2}(isone) \circ \ iota \ \ \textbf{where}\, isone\ x \ = \ (x = 1)$$

Optimiser: $\mathcal{O}(<, f_1, f_2, S)$ optimises the multiset according to a particular criterion (expressed through the ordering $<$) while preserving the structure of the multiset (described by the relation S). Consider for example the sorting of a sequence $(x_1, ..., x_n)$. The sequence is represented as a multiset of pairs (i, x_i) and the program proceeds by exchanging ill-ordered values:

$$\begin{aligned} & sort \ = \ \mathcal{O}(\ll, f_1, f_2, S) \quad \textbf{where} \\ & \quad f_1((i, a), (j, b)) \ = \ (i, b), \quad f_2((i, a), (j, b)) \ = \ (j, a) \\ & \quad ((i, a), (j, b)) \ \ll \ ((i', a'), (j', b')) \ \equiv (b \ < \ b') \\ & \quad S((i, a), (j, b)) \ = \ (i \ > \ j) \end{aligned}$$

Let us now take a few examples involving various combinations of tropes to suggest the transformations that we will study in the rest of the paper. The *primes* program defined above can be used to compute the prime factorization of a natural number.

$$pf(n) \ = \ (gen \ + \ del) \ \circ \ nt \ \circ \ int \ \circ \ primes(n)$$

$gen = \mathcal{E}(C, f_1, f_2)$ **where**	$del = \mathcal{S}_{1,2}(C)$ **where**	$int = \mathcal{T}(C, f)$ **where**
$C((x, y, k)) \ = \ (k \ > \ 0)$	$C((x, y, k)) \ = \ (k \ = \ 0)$	$C(x) \ = \ integer(x)$
$f_1((x, y, k)) \ = \ (x, y, k - 1)$		$f(x) \ = \ (x, n, 0)$
$f_2((x, y, k)) \ = \ x$		

The transmuter nt defined earlier in this section computes the number of times n is a multiple of each prime number. The transmuter int acts as an interface between $primes$ and nt, transforming each prime number x into a triple $(x, n, 0)$. The parallel combination $gen + del$ decomposes each triple (x, y, k) into k occurrences of x. For example if $n = 2^3 * 3 * 11^2$, then $pf(n) = \{2, 2, 2, 3, 11, 11\}$. We will show in the next sections that $gen + del$ can be transformed into $del \circ gen$, which means that the deletion of unnecessary elements can be postponed until the end of the computation. We also have that $nt \circ int \sim nt + int$ showing that both transmuters can be executed concurrently.

As a final illustration of the use of tropes, let us consider an image processing application: the edge detection problem [Sed88]. Each point of the image is originally associated with a grey intensity level. Then an intensity gradient is computed at each point and edges are defined as the points where the gradient is greater than a given threshold T. The gradient at a point is computed relative to its neighbours: only points at a distance d less than D are considered for the computation of the gradient. The gradient at a point is defined in the following way:

$$\begin{array}{l} G(P) = maximum(neighbourhood) - minimum(neighbourhood) \\ \textbf{where } neighbourhood = \{intensity(P') \mid distance(P, P') < D\} \end{array}$$

We use a multiset of quadruples (P, l, min, max). P is the coordinates of the point, l is its intensity level and min and max are the current values of respectively $minimum(neighbourhood)$ and $maximum(neighbourhood)$. The initial value of min and max is l. The evaluation consists in decreasing min and increasing max until the limit values are reached. This is achieved by two optimisers $decmin$ and $incmax$. The selector $disc$ discards the points where the gradient is less than the threshold and the transmuter rf removes unnecessary fields from the remaining elements.

$$\begin{array}{l} edges(n) = (disc + rf) \circ (decmin + incmax) \\ decmin = \mathcal{O}(\ll, f_1, f_2, S) \;\; \textbf{where} \\ \quad f_1(X, Y) = (X.P, X.l, Y.l, X.max), \quad f_2(X, Y) = Y \\ \quad (X', Y') \ll (X, Y) \equiv (X'.min < X.min) \\ \quad S(X, Y) = distance(X.P, Y.P) < D \\ incmax = \mathcal{O}(\prec, f_1, f_2, S) \;\; \textbf{where} \\ \quad f_1(X, Y) = (X.P, X.l, X.min, Y.l), \quad f_2(X, Y) = Y \\ \quad (X', Y') \prec (X, Y) \equiv (X'.max > X.max) \\ \quad S(X, Y) = distance(X.P, Y.P) < D \\ disc = \mathcal{S}_{1,2}(C) \;\; \textbf{where} \\ \quad C(X) = quadruple(X) \; and \; ((X.max - X.min) < T) \\ rf = \mathcal{T}(C, f) \;\; \textbf{where} \\ \quad C(X) = quadruple(X) \; and \; ((X.max - X.min) \geq T) \\ \quad f(X) = (X.P, X.l) \end{array}$$

The algebra of programs developed in the rest of the paper allows us to perform the following transformations:

$$\begin{array}{l} disc\ +\ rf\ \Longrightarrow\ disc\ \circ\ rf \\ disc\ +\ rf\ \Longrightarrow\ rf\ \circ\ disc \\ decmin\ +\ incmax\ \Longrightarrow\ decmin\ \circ\ incmax \\ decmin\ +\ incmax\ \Longrightarrow\ incmax\ \circ\ decmin \end{array}$$

These transformations can be used to tune a program to a particular architecture by increasing or decreasing its potential for parallelism. In the next section we provide a formal account of the semantics of tropes. Then we come back to the transformations suggested above and present their conditions of application.

3 Semantics of Combining Forms for Tropes

In this section we consider the operational semantics of programs consisting of tropes, together with the two *combining forms* introduced in the last section: sequential composition, $P_1 \circ P_2$, and parallel combination, $P_1 + P_2$

$$P \in \text{Programs} ::= \text{Tropes} \mid P \circ P \mid P + P$$

To define the semantics for these programs we define a single step transition relation between *configurations*. The terminal configurations are just multisets, and the intermediate configurations are program/multiset pairs written $\langle P, M \rangle$.

We define the single step transitions first for the individual tropes. For each instance of the tropes $t : \bar{x} \to A\bar{x} \Leftarrow R\bar{x}$ define

$$\langle t, M \rangle \to \begin{cases} \langle t, M - \bar{a}\ \cup A\bar{a} \rangle & \text{if } \exists \bar{a} \subseteq M.R\bar{a} \\ M & \text{otherwise} \end{cases}$$

So a program consisting of a single trope terminates when there are no applicable reactions in the multiset. Since no program defined using sequential composition can terminate in one step, single step terminal transitions are only defined for programs not containing sequential composition; thus the remaining *terminal* transitions are defined by the following rule:

$$\frac{\langle P, M \rangle \to M \quad \langle Q, M \rangle \to M}{\langle P + Q, M \rangle \to M}$$

The remaining transitions are defined using the concept of *active contexts*. An *active context*, $\mathbf{A}$ is a term containing a single hole [ ]:

$$\mathbf{A} ::= [\,] \mid P + \mathbf{A} \mid \mathbf{A} + P \mid P \circ \mathbf{A}$$

Let $\mathbf{A}[P]$ denote active context $\mathbf{A}$ with program P in place of the hole.

The idea of active contexts is that they isolate parts of a program that can affect the next transition, so that for example, the left-hand side of a sequential composition is not active (but it can become active once the right-hand side has terminated). The remaining transition rules are defined by the following two rules:

$$\frac{\langle t, M\rangle \to \langle t, M'\rangle}{\langle \mathbf{A}[t], M\rangle \to \langle \mathbf{A}[t], M'\rangle} \qquad \frac{\langle Q, M\rangle \to M}{\langle \mathbf{A}[P \circ Q], M\rangle \to \langle \mathbf{A}[P], M\rangle}$$

The first says that if there is a possible reaction in an active context then it can proceed, while the second says that if the right hand side of a sequential composition can terminate, then it can be erased. As usual, we use $\to^*$ to represent the reflexive and transitive closure of $\to$. Notice that since a program does not uniquely factor into a sub-program in an active context, the one-step transition relation is not deterministic, but it should not be difficult to see that it is total on non-terminal configurations (under the assumption that the transition relation on single tropes is total).

Inspection of the semantics shows that "programs" are not fixed. This leads us to the notion of the *residual* part of a program – the program component of any configuration that is an immediate predecessor of a terminal configuration (multiset). The *residual part* of a program P, written $\underline{\underline{P}}$, is defined by induction on the syntax:

$$\underline{\underline{t}} = t \qquad \underline{\underline{P_1 \circ P_2}} = \underline{\underline{P_1}} \qquad \underline{\underline{P_1 + P_2}} = \underline{\underline{P_1}} + \underline{\underline{P_2}}$$

This provides us with a simple (ie. weak) postcondition for programs. We define a predicate Φ on a program and a multiset to be true if and only if the residual part of the program is terminated with respect to the multiset.

Definition 1 (The postcondition Φ) $\Phi(P, M) \Leftrightarrow \langle \underline{\underline{P}}, M\rangle \to M$

Intuitively, $\Phi(P, M)$ holds if M is a possible result for the program P. The significance of this is that the predicate $\Phi(P, \_)$ can be constructed syntactically by considering (the negations of) the reaction conditions in $\underline{\underline{P}}$.

We can now define an ordering on programs. Intuitively, $P_1 \sqsubseteq P_2$ whenever, for each possible input M, if P_1 can diverge (i.e. rewrite forever, written $\langle P_1, M\rangle\uparrow$) then so can P_2, and if P_1 can terminate producing some multiset N then so can P_2.

Definition 2

$$\begin{aligned} P_1 \sqsubseteq P_2 \Leftrightarrow \forall M.((&\langle P_1, M\rangle\uparrow \Rightarrow \langle P_2, M\rangle\uparrow) \\ &\wedge (\forall N. \langle P_1, M\rangle \to^* N \Rightarrow \langle P_2, M\rangle \to^* N)) \end{aligned}$$

We write $P \sim Q$ if $P \sqsubseteq Q$ and $Q \sqsubseteq P$.

Without further ado, we present a key result from [HMS92] which establishes when sequential composition correctly implements parallel combination.

Theorem 1. $(\forall M.(\Phi(Q, M) \wedge \langle P, M\rangle \to^* N) \Rightarrow \Phi(Q, N)) \Rightarrow P \circ Q \sqsubseteq P + Q$

This theorem can be used to remove parallel combinations, replacing them by sequential compositions. The reverse transformation is only applicable under more stringent conditions. Given that $P \circ Q \sqsubseteq P + Q$, then $P \circ Q$ is a correct "interleaving" of the steps involved in executing $P + Q$; if we know that $P + Q$ always terminates (is *strongly normalising*, written $(P + Q) \downarrow_{must}$) and that it is confluent (deterministic) then all interleavings have the same effect and $P + Q \sqsubseteq P \circ Q$. This is the second key result from [HMS92] which allows us to parallelise sequential programs. Before presenting the theorem, we formally define confluence.

Definition 3 *P is* confluent*, written $Con(P)$, iff $\forall M.Con(P, M)$ where Con(P,M) is the predicate:*

$$\langle P, M\rangle \rightarrow^* \langle P_1, M_1\rangle \wedge \langle P, M\rangle \rightarrow^* \langle P_2, M_2\rangle \Rightarrow \\ \exists\langle P_3, M_3\rangle.(\langle P_1, M_1\rangle \rightarrow^* \langle P_3, M_3\rangle \wedge \langle P_2, M_2\rangle \rightarrow^* \langle P_3, M_3\rangle)$$

Theorem 2. $(P \circ Q \sqsubseteq P+Q \wedge Con(P+Q) \wedge (P+Q) \downarrow_{must}) \Rightarrow P+Q \sim P \circ Q$

We close this section which a theorem that collects together a number of properties first presented in [HMS92].

Theorem 3. *1.* $+$ *is associative and commutative,* $\circ$ *is assoc.,* $\circ$ *is monotone.*
2. $(P_1 + P_3) \circ P_2 \sqsubseteq (P_1 \circ P_2) + P_3$
3. $(P_1 + \underline{\underline{P_3}}) \circ (P_2 + P_3) \sqsubseteq (P_1 \circ P_2) + P_3$
4. $(P \sqsubseteq \overline{P} + P) \wedge (P \sim \underline{\underline{P}} \Rightarrow P \sim P + P)$
5. $(P_1 + P_2) \circ (Q_1 + Q_2) \sqsubseteq (P_1 \circ Q_1) + (P_2 \circ Q_2)$
6. $P_1 \circ (P_2 + P_3) \sqsubseteq (P_1 \circ P_2) + (P_1 \circ P_3)$

4 Properties

We show in this section how some general results, including some of those presented above, can be specialised to derive useful properties of tropes. We first study the conditions under which a program is confluent and terminating. Then we prove a number of laws concerning compositions of tropes.

4.1 General Confluence and Termination Properties

The computation model of Gamma is closely related to conditional term rewriting. We call a program *simple* if it does not contain any sequential compositions. One property of the semantics defined earlier is that for any simple program P, if $\langle P, M\rangle \rightarrow \langle Q, N\rangle$ then $P = Q$. Consequently, a simple program P can be viewed as defining a multiset rewriting relation which is an associative, commutative, conditional rewriting system. Arbitrary programs may not be viewed in this way because of the combinations of $+$ and $\circ$ that are allowed. There are, however, some general results for Abstract Reduction Systems [Klo90] which are useful.

For simple programs confluence may be proved using one of the standard decompositions, such as *Newman's Lemma* (see eg. [Klo90]). A most useful method for proving the termination of Gamma programs (see [BM90]) is the Dershowitz-Manna multiset ordering [DM79] in which a well-founded multiset ordering can be derived from a well-founded ordering on the elements of the multiset.

We now consider how these properties interact with the combinators that we have introduced. It is easy to verify that the sequential composition of two programs which are confluent (strongly normalising) will result in a combined program which is confluent (resp. strongly normalising). Now we consider programs of the form $P_1 + P_2$ where P_1 and P_2 are simple. We call the multiset-rewriting relations associated with the two sub-programs $\rightarrow^*{}_1$ and $\rightarrow^*{}_2$, respectively, and we will say that they *commute* if they satisfy a diamond property of the following form:

$$\exists M_3.\, M_1 \;{}_1^*\!\!\leftarrow M \rightarrow_2^* M_2 \Longrightarrow M_1 \rightarrow_2^* M_3 \;{}_1^*\!\!\leftarrow M_2.$$

Now if $\rightarrow^*{}_1$ and $\rightarrow^*{}_2$ are individually confluent and they commute, then the combined rule system is also confluent; this a consequence of the *Hindley-Rosen Lemma* which is proved by a simple diagram chase. Termination, however is not necessarily preserved - we require a stronger condition than commutativity. If we add the condition that $\rightarrow_2$-rewrites cannot create $\rightarrow_1$-redexes, then termination of the combined rule system follows from the separate termination of the two programs.

The generalisation of these last two paragraphs to arbitrary programs constructed from the combinators is a matter for further investigation.

4.2 Confluence and Termination of Tropes

Transmuter: For termination we require a well-founded ordering, $<$, on the multiset such that: $C(x) \Rightarrow f(x) < x$. Since a transmuter only selects one element at a time, it is trivially confluent.

Reducer: Termination follows trivially from cardinality considerations. In turning to confluence, we consider the reducer $\mathcal{R}(C, f)$. We require the following notion: the symmetric predicate C is *f-preserved* iff:

$C(x, y) \wedge C(x, z) \Rightarrow C(f(x, y), z) \wedge C(y, f(x, z))$.

A sufficient condition for a reducer to be confluent is that f is associative and commutative and that C is f-preserved.

Optimiser: The detailed consideration of the optimiser is omitted from this paper due to lack of space. It has proved difficult to derive general conditions for termination and confluence of this trope but we have identified two special cases which are useful in program development. These will be described elsewhere.

Expander: We consider the expander, $\mathcal{E}(C, f_1, f_2)$ applied to a multiset S. Define S_C to be the subset of S whose elements satisfy the predicate C. Suppose that we have a well-founded ordering, $<$, on S_C. We say that a function, f, is *reductive* on S_C if for all $x \in S_C$, either $f(x) \notin S_C$ or $f(x) < x$. A sufficient condition for termination of an expander is that the two functions f_1 and f_2 are

reductive on S_C. Since an expander has a single variable on the left hand side, there are no overlapping redexes and confluence is trivial.
Selector: Termination of this rule follows trivially by cardinality considerations. Given the generality of the rule, it is difficult to give conditions for confluence. Restricting ourselves to rules of the form $\mathcal{S}_{2,2}(C)$, a sufficient condition is that C is antisymmetric and transitive.

4.3 Laws

We start with some definitions of derived relations which will provide us with some useful notations. Given R, a reaction condition of arity n, we define:

$\overline{R}_M(x) \Leftrightarrow \forall \bar{a} \subseteq M.\, x \in \bar{a} \Rightarrow \neg R(\bar{a})$

Informally, $\overline{R}_M(x)$ says that element x is unable to partake in a reaction within multiset M. If this holds for arbitrary M, we just write $\overline{R}(x)$, with the intuition that x cannot partake in any reaction for which R is the associated reaction condition. We say that two reaction conditions are *exclusive* if the sets of elements that satisfy them are guaranteed to be disjoint:

$Exclusive(R, R') \Leftrightarrow \forall x.(\overline{R}(x) \vee \overline{R'}(x))$

We can now start to list some properties of tropes. The first set are general laws, expressed in terms of individual reactions written abstractly as condition/action pairs (R, A):

Theorem 4. *1. $Exclusive(R, R')$ and $(\forall a \in A(x_1, \ldots, x_n).\overline{R'}(a))$ implies $(R, A) + (R', A') \sim (R, A) \circ (R', A')$*
2. If $\langle (R, A), M\rangle \rightarrow \langle (R, A), N\rangle$ and $\forall x \in M\backslash N.\, \overline{R'}_M(x)$ imply that[4] $\forall y \in N\backslash M.\, \overline{R'}_N(y)$, then $(R, A) + (R', A') \sqsupseteq (R, A) \circ (R', A')$

The first part is quite intuitive, since the condition expresses complete interference-freedom of R and R'. The second part has a somewhat stronger precondition which can be informally read as: if all the removed elements of a (R, A)-reaction were R'-stable for the original multiset then all the new elements will be stable for the resulting multiset. This is sufficient to guarantee that after termination of (R', A') subsequent reactions by (R, A) could not generate any elements that enable R', and hence the refinement.

The next theorem states some properties of particular tropes and their interactions obtained by specialising the above properties:

Theorem 5. *1. $(C'(x) \wedge \overline{C}(x) \Rightarrow \overline{C}(f'x)) \Rightarrow$
$\mathcal{S}_{i,j}(C) + \mathcal{T}(C', f') \sqsupseteq \mathcal{T}(C', f') \circ \mathcal{S}_{i,j}(C)$*
2. $\mathcal{S}_{i,j}(C_1 \vee C_2) \sim \mathcal{S}_{i,j}(C_1) + \mathcal{S}_{i,j}(C_2)$
*3. $(\neg C(x) \wedge C'(x) \Rightarrow \neg C(f'(x))) \Rightarrow$
$\mathcal{T}(C', f') + \mathcal{E}(C, f_1, f_2) \sqsupseteq \mathcal{T}(C', f') \circ \mathcal{E}(C, f_1, f_2)$*
*4. $(\neg C'(x) \wedge C(x) \Rightarrow \neg C'(f_1(x)) \wedge \neg C'(f_2(x))) \Rightarrow$
$\mathcal{T}(C', f') + \mathcal{E}(C, f_1, f_2) \sqsupseteq \mathcal{E}(C, f_1, f_2) \circ \mathcal{T}(C', f')$*
5. $\forall P.\mathcal{S}_{i,j}(C) + P \sqsupseteq \mathcal{S}_{i,j}(C) \circ P$

[4] Notation: $M\backslash N$ is multiset difference, so, eg. $\{1, 1, 2\}\backslash\{1, 3\} = \{1, 2\}$.

Now we use the various properties to prove some of the transformations mentioned in Section 2.

Lemma 6. *1.* $gen + del \sim del \circ gen$
2. $nt + int \sim nt \circ int$
3. $disc + rf \sqsupseteq disc \circ rf$
4. $disc + rf \sqsupseteq rf \circ disc$
5. $decmin + incmax \sqsupseteq incmax \circ decmin$
6. $decmin + incmax \sqsupseteq decmin \circ incmax$

Proof. 1. Then we have that the two reaction conditions are exclusive, and therefore, since *del* has an empty action, we have the desired result by Theorem 4 (1).
2. Since the two tropes consume elements of different types, we have that their conditions are exclusive. Moreover, elements produced by *nt* are stable for *int*. Consequently the result follows by Theorem 4 (1).
3. Follows immediately from Theorem 5 (5).
4. Suppose that $(quadruple(X) \wedge (X.max - X.min) \geq T$. Then we have $\overline{C}(X)$ and $\overline{C}(f(X))$ thus, we have the required result by Theorem 5 (1).
5. and 6. follow by application of Theorem 4 (2). The verifications of the preconditions are straightforward.

5 Concluding Remarks

The programming style defined and studied in this paper stemmed from previous work on program construction in the Gamma formalism [BM93]. The main achievements of this paper are:

- The formal definition of the tropes and their semantics.
- The study of their algebra of programs.

A collection of examples illustrating the relevance of the programming style advocated in this paper can be found in [BM93].

Our main objectives are closely related to those of the Unity designers [CM88]. In contrast with their approach we have emphasized the development of a calculus of program transformation whereas they have focussed on logical aspects of program refinement. The use of the multiset as a basic data structure for parallel programming (and subsequent program transformation) has also been investigated in the functional programming setting by Roe [Roe91]. Conditional rewriting as a general model of parallel programming has been advocated by Meseguer [MW91] who shows how various computational formalisms can be expressed in this framework.

In [HMS92] we developed a formal semantics for a Gamma variant and studied its properties (some of which appear as Theorem 3). The weakness of our earlier work was that we did not specify the details of the primitive rewrites, the (R, A)-pairs; as a consequence our calculus was restricted to quite general transformations. In this paper we have introduced five primitive program forms, the tropes, as a basis for parallel programming. We have shown how many of the examples can be recast using the tropes and the two combining forms. Being specific about the primitive forms allows us to develop a much richer calculus; the new rules are summarised in Theorems 4 and 5. We have shown how these rules may be used to transform some of the earlier examples.

The multiset rewriting paradigm has also been applied in the context of reactive systems [MW91] [BM93]. Further work will include the study of the tropes in that context. The semantics of the tropes in section 3 involves two different kinds of rewritings: *active steps* which rewrite the multiset and *passive steps* during which the program is modified. The latter does not have any obvious analogy in the term rewriting approach. The work reported here will form the starting point for a deeper study of the similarities and differences with commutative and associative term rewriting systems.

Acknowlegement Thanks to T. Mogensen for comments on an earlier draft.

References

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[BM90] J.-P. Banâtre and D. Le Métayer. The Gamma model and its discipline of programming. *Science of Computer Programming*, 15:55–77, 1990.

[BM92] J.-P. Banâtre and D. Le Métayer, editors. *Research Directions in High-level Parallel Programming Languages*. Springer-Verlag, LNCS 574, 1992.

[BM93] J.-P. Banâtre and D. Le Métayer. Programming by multiset transformation. *CACM*, January 1993.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[DM79] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Comm. ACM*, 22:465–476, 1979.

[HMS92] C. Hankin, D. Le Métayer, and D. Sands. A calculus of Gamma programs. Research Report DOC 92/22, Department of Computing, Imperial College, 1992.

[Hug89] J. Hughes. Why functional programming matters. *The Computer Journal*, 2(32):98–107, April 1989.

[Klo90] J.W. Klop. Term rewriting systems. Technical Report CS-R9073, CWI, 1990.

[Maa92] A. Maasen. Parallel programming with data structures and higher-order functions. *Science of Computer Programming*, 18:1–38, 1992.

[MW91] J. Meseguer and T. Winkler. Parallel programming in Maude. In *[BM92]*, 1991.

[Roe91] P. Roe. *Parallel Programming using Functional Languages.* PhD thesis, Glasgow University, 1991.

[Sed88] R. Sedgewick. *Algorithms.* Addison-Wesley, 1988.