Theoretical Computer Science 192 (1998) 233–258

Theoretical Computer Science

Refining multiset transformers

Chris Hankin[a], Daniel Le Métayer[b], David Sands[c]

[a] *Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, UK*
[b] *IRISA, Campus Universitaire de Beaulieu, 35042-Rennes Cédex, France*
[c] *Department of Computing Science, Chalmers University of Technology and Göteborg University, S-412 96 Göteborg, Sweden*

Abstract

Gamma is a minimal language based on local multiset rewriting with an elegant chemical reaction metaphor. The virtues of this paradigm in terms of systematic program construction and design of parallel programs have been argued in previous papers. Gamma can also be seen as a notation for coordinating independent programs in a larger application. In this paper, we study a notion of refinement for programs involving parallel and sequential composition operators, and derive a number of programming laws. The calculus thus obtained is applied in the development of a generic "pipelining" transformation, which enables certain sequential compositions to be refined into parallel compositions.

Keywords: Gamma; Multiset rewriting; Program transformation

1. Introduction

We first describe the general motivation of the work presented here before summarising the main results developed in the body of the paper.

1.1. *Motivation*

The notion of sequential computation has played a central rôle in the design of most programming languages in the past. This state of affairs was justified by at least two good reasons:

- Sequential models of execution provide a good form of abstraction of algorithms matching the intuitive perception of a program defined as a "recipe" for preparing the desired result.
- Actual implementations of programs were made on single processor architectures, reflecting this abstract sequential view.

* Corresponding author. E-mail: clh@doc.ic.ac.uk.

However, the computer science landscape has evolved considerably since then. These changes have been caused by dramatic progress in the hardware technology and tremendous increase in the size of real software applications. Let us examine these two issues in turn and assess their impact on programming languages.

- We have seen in the last few years the widespread development of electronic networks made possible by the progress in communication technology. This trend is likely to be accelerated in the future. As a consequence, a computer can no longer be considered in isolation; it should rather be seen as a node in a graph representing a distributed system. Individual computers themselves are no longer single processors: parallelism is now integrated in various ways and at all levels of the computation (from low-level pipelining and superscalar processors to shared-memory multiprocessors and fine-grained parallel machines). Programming such machines obviously requires parallel languages and models of computation.
- As a result of the growing needs and the decreasing cost of hardware, we have seen a tremendous proliferation of software systems. This evolution introduces new problems: software developed through a long period of time tends to grow in size and complexity and become extremely difficult to understand and to maintain. The cost incurred by this complexity is becoming a serious concern and a major challenge today is to provide ways of organising software in order to make big applications manageable and to favour the reuse of existing products. Various languages have been proposed recently to tackle these problems: they are called software architecture languages [2], or coordination languages [10]. A key feature of these languages is to allow the description of interactions between individual pieces of software (which may themselves be written in different programming languages).

Thus, the situation created by this double evolution has placed new needs on the design of languages: sequentiality should no longer be seen as the prime programming paradigm but just as one of the possible forms of cooperation between individual entities.

The Gamma formalism presented a few years ago precisely captures the idea of considering parallelism as the basic program structuring facility. Gamma is a kernel language which can be introduced intuitively through the chemical reaction metaphor. The unique data structure in Gamma is the multiset which can be seen as a chemical solution. A simple program is a pair (*Reaction condition*, *Action*). Execution proceeds by replacing in the multiset elements satisfying the reaction condition by the products of the action. The result is obtained when a stable state is reached, that is to say when no more reaction can take place. The following is an example of a Gamma program computing the maximum element of a non-empty set.

$$max : x, y \to x \Leftarrow x \geqslant y$$

$x \geqslant y$ specifies a property to be satisfied by the selected elements x and y. These elements are replaced in the set by the value x. Nothing is said in this definition about the order of evaluation of the comparisons. If several disjoint pairs of elements satisfy the conditions, the comparisons and replacements can be performed in parallel.

Let us consider as another introductory example a sorting program. We use a set of pairs (*index*, *value*) and the program exchanges ill-ordered values until a stable state is reached and all values are well-ordered:

$$sort_A : (i,x),(j,y) \rightarrow (i,y),(j,x) \Leftarrow (i<j) \quad \text{and} \quad (y<x)$$

The interested reader may find in [6] a longer series of examples (string processing problems, graph problems, geometry problems, etc.) illustrating the Gamma style of programming. The possibility of getting rid of artificial sequentiality in Gamma has two important consequences:

- It makes Gamma suitable as an intermediate language in the program derivation process allowing the programmer to design a very abstract version of his program in the first place (which is easier to prove correct); this version is then specialised for the sake of efficiency by introducing extra control. The benefit of Gamma in systematic program construction is illustrated in [5].
- Gamma programs do not have any sequential bias and the language leads naturally to the construction of parallel programs. It also makes Gamma a potential candidate for a coordination language in which atomic actions would be seen as individual pieces of software using the multiset as the only cooperation facility (just as Linda promotes the design of coordination through a shared tuple space). The interested reader can find in [20] more details about the design of a coordination (or software architecture) language inspired by Gamma.

Furthermore, the very minimal nature of the language allows us to provide a clean and concise semantics that can be used to reason about programs. So we believe that Gamma is a promising starting point for the design of a language answering the questions raised above. However the basic version of Gamma mentioned so far suffers one major weakness: it lacks any means for structuring programs, or building complex programs from simple ones. For the sake of modularity, it is desirable that a language offers a rich set of operators for combining programs. It is also fundamental that these operators enjoy a useful collection of algebraic laws in order to make it possible to reason about programs.

The essence of this paper is the presentation of a set of operators for Gamma and a study of their semantics and the corresponding calculus of programs. We put emphasis on one program transformation called "pipelining" which allows us to transform sequential composition of programs (which may be the most natural way to build complex programs from existing ones) into parallel compositions (which may be more efficient).

In the rest of this section, we sketch the main themes of this paper: we introduce informally a sequential operator and a parallel operator; then we provide some intuition about the operational semantics of this enriched Gamma language and the algebra of programs it gives rise to. The pipelining transformation is introduced and motivated. To conclude this section, we recall the definition of multisets and their operators, which are central to the technical developments of this paper.

1.2. Sequential and parallel composition operators

The basic operators that we consider in this paper are the sequential composition $P_1 \circ P_2$ and the parallel composition $P_1 \mid P_2$. The intuition behind $P_1 \circ P_2$ is that the stable multiset reached after the execution of P_2 is given as argument to P_1. On the other hand, the result of $P_1 \mid P_2$ is obtained (roughly speaking) by executing the reactions of P_1 and P_2 (in any order, possibly in parallel), terminating only when neither can proceed further. The termination condition is particularly significant and heavily influences our choice of semantics for parallel composition. As an example of sequential composition of Gamma programs, let us consider another version of sort:

$$\begin{aligned} &sort_B : match \circ init \\ &\text{where} \quad init : (x \rightarrow (0,x) \Leftarrow integer(x)) \\ &\qquad match : ((i,x),(i,y) \rightarrow (i,x),(i+1,y) \Leftarrow x \leqslant y) \end{aligned}$$

The program $sort_B$ takes a multiset of integers and returns an increasing list encoded as a multiset of pairs $(index, value)$. The reaction *init* gives each integer an initial rank of zero. When this has been completed, *match* takes any two elements of the same rank and increases the rank of the larger.

The case for parallel composition is slightly more involved. In fact $sort_B$ could have been defined as well as

$$sort_B : match \mid init$$

because the reactions of *match* can be executed in parallel with the reactions of *init*. As far as the semantics of parallel composition is concerned, the key point is that we need a *synchronised termination* of P_1 and P_2 for $P_1 \mid P_2$ to terminate. It may be the case that at some stage of the computation none of the reaction conditions of, P_1 (resp. P_2) holds; but some reactions by P_2 (resp. P_1) may create new values which will then be able to take part in reactions by P_1 (resp. P_2). This situation precisely occurs in the above example where no reaction of *match* can take place in the initial multiset; but *init* transforms the multiset and triggers subsequent reactions by *match*. Thus, the termination condition of $P_1 \mid P_2$ indicates that neither P_1 nor P_2 can terminate unless both terminate. This contrasts with the *asynchronous termination* condition of most process calculi (where if P_1 terminates (reduces to *nil*) then $P_1 \parallel P_2 \rightarrow P_2$). Gamma programs should rather be compared with rewriting systems, and their parallel composition with the union of rewriting systems. In this context, it is natural to say that a normal form is reached only when none of the systems possess a rule which can apply to the term.

1.3. An operational semantics and an algebra of programs

In Section 2, we propose an operational semantics of an enhanced version of Gamma with sequential and parallel composition. We derive a rich set of program refinement and equivalence laws for parallel and sequential composition. So, for example, the input–output behaviour of $sort_B$ is equivalent to that of program $sort_A \mid sort_B$. This is (by definition) $sort_A \mid (match \circ init)$, and this is refined by the program

$$(sort_A \mid match) \circ init.$$

This refinement is an instance of a general refinement law:

$$(P \mid Q) \circ R \leqslant P \mid (Q \circ R)$$

We particularly focus on conditions under which $P_1 \circ P_2$ can be transformed into $P_1 \mid P_2$ and vice versa. These transformations are useful to improve the efficiency of a program with respect to some particular machine and implementation strategy. Let us take another example [6] to illustrate this point:

$$connected = singleton \circ (P_1 \mid P_2)$$

$$where$$

$$P_1 : v, w, (m,n) \to v \cup w \Leftarrow nodes(v) \wedge nodes(w) \wedge m \in v \wedge n \in w$$

$$P_2 : v, (m,n) \to v \Leftarrow nodes(v) \wedge m \in v \wedge n \in v$$

This program is used to detect whether a graph is strongly connected or not. The initial multiset representation of the graph consists of the collection of singleton sets of *nodes*, together with the collection of *edges*. A pair (m,n) is used to represent an edge linking nodes m and n. It proceeds by building bigger and bigger aggregates of connected nodes (through P_1). The predicate "nodes" simply allows the reactions to distinguish between an edge and a node set. P_2 is used to remove edges connecting two nodes belonging to the same set. Once this process has stabilised, the graph is connected if all the nodes have been gathered into a single set. This is tested via the primitive *singleton* – not specified here.

Our algebra of programs allows us to show, for example, that $P_1 \mid P_2$ is equivalent to $P_2 \circ P_1$ which means that all the reactions of P_2 can be postponed until no more P_1 reactions can take place. If the target architecture is a sequential one (or even a parallel one with relatively few processors) $P_2 \circ P_1$ will be more efficient because many useless tests of the reaction condition of P_2 will be avoided; however, $P_1 \mid P_2$ might turn out to be a better version if executed on a massively parallel machine because unnecessary edges can be removed by P_2 at the same time as aggregates are built by P_1.

In this paper we focus on one program transformation called "pipelining" which allows us to transform sequential compositions of programs into parallel compositions. The significance of this technique comes from the fact that sequential composition is often the natural way to build complex programs from existing ones. The pipeline program obtained as a result of the transformation connects the subtasks in such a way that the output of one task feeds piecemeal into the input of the next. The transformation is based on a notion of *stable elements* which cannot partake in any reaction of a given program. A sequential composition $P_2 \circ P_1$ can be transformed into $P_2 \mid P_1$ (in order to allow P_2 to consume the stable elements of P_1 as soon as they are produced) provided that P_2 does not interfere with the unstable elements of P_1. This is achieved by adding, in parallel, an *interface* program which tags stable elements of P_1 and modifying P_2 so that it can only operate on tagged data. Section 3 presents the pipelining transformation in detail with an example illustrating its relevance.

1.4. Multisets

A *multiset*, sometimes called a *bag*, is a set-like collection in which elements may be duplicated. So, for example:

$$\{1,1,2,2,2,2,3,4,5,5\}$$

is a valid multiset. It is sometimes convenient to think of a multiset, M, over a set, X, as a function, $M : X \rightarrow Nat$, which maps each element to its *multiplicity* – the number of times the element occurs in the multiset. We write $|M|$ for the set of elements in M.

Given multisets, M and N, we write $M \backslash N$ for the multiset difference:

$$M\backslash N(x) = max(0, M(x) - N(x))$$

and we write $M \uplus N$ for multiset join:

$$M \uplus N(x) = M(x) + N(x).$$

2. Operational semantics of Gamma programs

In this section we consider the operational semantics of programs consisting of basic reactions (written $A \Leftarrow R$, where R is the reaction condition, and A is the associated action, both assumed to have the same arity), together with two *combining forms*: sequential composition, $P_1 \circ P_2$, and parallel combination, $P_1 \mid P_2$ as introduced in [18].

$$\begin{aligned} P \in \mathbf{P} ::= \; & (A \Leftarrow R) \mid P \circ P \mid \\ & (P \mid P) \end{aligned}$$

For the purposes of this paper, we will consider A to be a function and R to be a predicate in first-order logic. We write $\mathbf{M}$ to denote the set of finite multisets of elements. The domain of the elements is left unspecified, but is expected to include integers, booleans, and closed under products. 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 $\langle P, M\rangle$, where $M \in \mathbf{M}$. We will often use the alternative syntax for basic programs (reactions) as was already done in the introduction:

$$G : x_1, \ldots, x_n \rightarrow A(x_1, \ldots, x_n) \;\Leftarrow\; R(x_1, \ldots, x_n)$$

for: $G : (A \Leftarrow R)$, where R and A are of arity n.

$$\langle (A \Leftarrow R), M\rangle \rightarrow ((A \Leftarrow R), (M \setminus \{\!|a_1 \dots a_n|\!\} \uplus A(a_1 \dots a_n))$$
$$\text{if } a_1 \dots a_n \in M \text{ and } R(a_1 \dots a_n)$$

$$\langle (A \Leftarrow R), M\rangle \rightarrow M \text{ if } \neg\exists a_1 \dots a_n \in M \,.\, R(a_1 \dots a_n)$$

$$\frac{\langle P_2, M\rangle \rightarrow M}{\langle P_1 \circ P_2, M\rangle \rightarrow \langle P_1, M\rangle} \qquad \frac{\langle P_2, M\rangle \rightarrow \langle P_2', M'\rangle}{\langle P_1 \circ P_2, M\rangle \rightarrow \langle P_1 \circ P_2', M'\rangle}$$

$$\frac{\langle P_1, M\rangle \rightarrow M \quad \langle P_2, M\rangle \rightarrow M}{\langle P_1 \mid P_2, M\rangle \rightarrow M}$$

$$\frac{\langle P_1, M\rangle \rightarrow \langle P_1', M'\rangle}{\langle P_1 \mid P_2, M\rangle \rightarrow \langle P_1' \mid P_2, M'\rangle} \qquad \frac{\langle P_2, M\rangle \rightarrow \langle P_2', M'\rangle}{\langle P_1 \mid P_2, M\rangle \rightarrow \langle P_1 \mid P_2', M'\rangle}$$

Fig. 1. Structural operational semantics of Gamma.

The semantics of Gamma programs is given in Fig. 1 in the standard *structural operational semantics* style. We make the assumption that the predicate R in a reaction condition is a total function from tuples of multiset elements to the truth values, and that the action function A is total on the domain of R. From these assumptions it easily follows that the one step evaluation relation is total, i.e. that for all nonterminal configurations $\langle P, M\rangle$ there is at least one configuration U (either terminal or nonterminal) such that $\langle P, M\rangle \rightarrow U$.

Given this basic transition relation for programs, we now consider orderings on programs according to their operational behaviours.

2.1. Relational orderings

A number of "refinement" orderings on programs arise from the various natural ways to compare programs on the basis of their input–output (or relational) behaviour. One possible "behaviour" which we should consider significant is the possibility of nontermination for a given input. Nontermination, or "divergence" is a predicate on program configurations:

Definition 1. P may diverge on M, $\langle P, M\rangle\uparrow$, if there exist $\{\langle P_i, M_i\rangle\}_{i\in\omega}$ such that $\langle P_0, M_0\rangle = \langle P, M\rangle$ and $\langle P_i, M_i\rangle \rightarrow \langle P_{i+1}, M_{i+1}\rangle$.

It is convenient to abstract the possible relational behaviours of a program as a set of possible input–output pairs. This includes the possibility of non-termination, which we represent as a possible "output" using symbol "$\perp$":

Definition 2. The behaviours of a program P, $\mathscr{B}(P) \subset \mathbf{M} \times (\mathbf{M} \cup \{\perp\})$ is defined as

$$\mathscr{B}(P) = \{(M, N) \mid \langle P, M\rangle \rightarrow^* N\} \cup \{(M, \perp) \mid \langle P, M\rangle\uparrow\}$$

This definition of behaviours is aimed at a study of relational properties rather than reactive properties. The development in this section carries over to the case of reactive behaviours (e.g. traces) but one might argue that different composition operators should be considered from the outset (e.g. recursion).

Note that every program has *some* behaviours, since we assume the reactions and actions are total. In fact, this assumption gives us a much stronger property: for every P, M, either $(M,N)\in\mathscr{B}(P)$ for some N, or $(M,\perp)\in\mathscr{B}(P)$ (or both).

In [18] a variety of orderings on programs was defined, based on their behaviours, by considering the associated discrete power-domain orderings on $\mathbf{M}_\perp$. In this study we only consider the "relational ordering" (the $\leqslant_R$ order of [18]).

Definition 3. $P\leqslant Q$ if and only if $\mathscr{B}(P)\subseteq\mathscr{B}(Q)$.

We read $P\leqslant Q$ as "P correctly implements Q" or "P refines[1] Q", since if any result (including possible nontermination) is considered acceptable from Q, then any behaviour that P can exhibit must also be acceptable. This view of refinement fits with an implementation of "loose" nondeterminism, in which we assume that an implementation must be able to realise some but not necessarily *every* behaviour of a program. Let $\equiv$ be the associated behavioural equivalence, so $P\equiv Q$ if $P\leqslant Q$ and $Q\leqslant P$.

Fact 4. *$\equiv$ is a congruence with respect to $\circ$, that is*
1. $P_1\equiv P_2 \Rightarrow P\circ P_1\equiv P\circ P_2$,
2. $P_1\equiv P_2 \Rightarrow P_1\circ P\equiv P_2\circ P$.

However, $\equiv$ is not a congruence with respect to $|$. Consider the following three rules:

$$P:\ (n,m\rightarrow n+m)$$

$$Q:\ (n\rightarrow n-1,1\Leftarrow n>1)$$

$$R:\ (n\rightarrow n+1\Leftarrow n<10)$$

Then $P\circ Q\equiv P$ but $(P\circ Q)\,|\,R\not\equiv P\circ R$; the left hand side may diverge because of the interaction between Q and R, whereas the right hand side does not have this possibility. We will return to this issue in the Conclusion.

2.2. Constrained refinement

It is often difficult to refine programs without imposing some constraints on the nature of the data upon which the program is executed. For present purposes we introduce an

[1] This is consistent with the usual notion of refinement, but the comparison symbol $\leqslant$ is used in the opposite direction.

indexed variant of the operational ordering, which expresses refinement with respect to a simple local precondition on the elements of the multisets.

Definition 5. For any set of elements E, let $Multi(E)$ denote the set of all finite multisets of elements in E.

Now define the constrained behaviours of a program,

$$\begin{aligned}\mathscr{B}(P)_E = \{(M,N) \mid M \in Multi(E); N \in \mathbf{M}; \langle P,M\rangle \rightarrow^* N\} \\ \cup \{(M,\bot) \mid M \in Multi(E); \langle P,M\rangle\uparrow\}\end{aligned}$$

Finally, the constrained refinement relations for each set of elements E are defined by

$$P \leqslant_E Q \quad \text{if and only if} \quad \mathscr{B}(P)_E \subseteq \mathscr{B}(Q)_E.$$

Constrained equivalence, $\equiv_E$ is defined in the obvious way. The constraint on inputs is *local* in the sense that it imposes a restriction on the individual elements appearing in a multiset, and not an arbitrary precondition on the whole multiset.

2.3. *The residual program*

The operational rules for sequential composition imply that the program component of a configuration is not static during computation. But the possible ways in which it can change are limited by the structure of the program (and not the multiset). In particular, 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).

Definition 6. The *residual part* of a program P, written $\underline{\underline{P}}$, is defined by induction on the syntax:

$$\begin{aligned}\underline{\underline{(A \Leftarrow R)}} &= (A \Leftarrow R)\\ \underline{\underline{P_1 \circ P_2}} &= \underline{\underline{P_1}}\\ \underline{\underline{P_1 \mid P_2}} &= \underline{\underline{P_1}} \mid \underline{\underline{P_2}}\end{aligned}$$

We will say that a program is *simple* if it does not contain any sequential compositions. Note that the range of the mapping $\underline{\underline{\cdot}}$ is the set of simple programs, and if P is simple then $P = \underline{\underline{P}}$.

Proposition 7. $\langle P,M\rangle \rightarrow^* N \Leftrightarrow \langle P,M\rangle \rightarrow^* \langle \underline{\underline{P}},N\rangle \rightarrow N$.

Proof. ($\Rightarrow$) Follows from a straightforward induction on the length of the derivation of $\langle P,M\rangle \rightarrow^* N$.

($\Leftarrow$) Immediate because $\rightarrow^*$ is the transitive closure of $\rightarrow$. $\square$

Knowledge of the syntactic form of the program part of a configuration just before the termination step provides us with a simple (i.e. weak) postcondition for programs. We define a predicate Φ on a program and multiset to be true if and only if the residual part of the program is terminated with respect to the multiset.

Definition 8 (*The postcondition* Φ). $\Phi\langle P,M\rangle \Leftrightarrow \langle \underline{\underline{P}},M\rangle \to M$.

Intuitively, $\Phi\langle P,M\rangle$ holds if M is a possible result for the program P (as determined by the reaction conditions in the residual program), i.e. if $\langle P,M\rangle \to^* N$ then $\Phi\langle P,N\rangle$. $\Phi\langle P,M\rangle$ can be constructed syntactically by considering (the negations of) the reaction conditions in $\underline{\underline{P}}$:

$$\Phi\langle (A \Leftarrow R),M\rangle = \forall x_1,\dots,x_n \in M.\neg R(x_1,\dots,x_n)$$
$$\Phi\langle P \circ Q,M\rangle = \Phi\langle \underline{\underline{P}},M\rangle$$
$$\Phi\langle P \mid Q,M\rangle = \Phi\langle \underline{\underline{P}},M\rangle \wedge \Phi\langle \underline{\underline{Q}},M\rangle$$

For example, for the program $sort_A$ in the introduction, by negating the reaction condition we obtain

$$\Phi\langle sort_A,M\rangle \Leftrightarrow \forall \{\!|(i,x),(j,y)|\!\} \subseteq M, \quad i<j \Rightarrow x \leqslant y,$$

i.e. an element with a higher index has at least as large a value.

We close this section with a theorem which gives a condition under which sequential composition refines parallel composition. In order to prove the theorem, we require two lemmas. The first provides a factorisation for the derivation sequence associated with a program defined by sequential composition:

Lemma 9. $\langle P \circ Q,M\rangle \to^* N \Leftrightarrow \exists N'.(\langle Q,M\rangle \to^* N' \wedge \langle P,N'\rangle \to^* N)$.

Proof. ($\Rightarrow$)

$$\langle P \circ Q,M\rangle \to^k N \Rightarrow \exists N'.(\langle Q,M\rangle \to^{k_1} N' \wedge \langle P,N'\rangle \to^{k_2} N)$$

with $k = k_1 + k_2$ follows from a straightforward induction on k.

($\Leftarrow$) $\langle Q,M\rangle \to^* N'$ implies $\langle Q,M\rangle \to^* \langle \underline{\underline{Q}},N'\rangle$ and $\langle \underline{\underline{Q}},N'\rangle \to N'$, by Proposition 7. Thus,

$$\langle P \circ Q,M\rangle \to^* \langle P \circ \underline{\underline{Q}},N'\rangle \to \langle P,N'\rangle$$

and the result follows using the second conjunct. □

The second lemma relates derivations for sequential and parallel composition.

Lemma 10. $\langle P \circ Q,M\rangle \to^* N \Rightarrow \langle P \mid Q,M\rangle \to^* \langle \underline{\underline{P}} \mid \underline{\underline{Q}},N\rangle$.

Proof. By Lemma 9, there is a N' with $\langle Q,M\rangle \to^* N'$ and $\langle P,N'\rangle \to^* N$. Thus, by Proposition 7, $\langle Q,M\rangle \to^* \langle \underline{\underline{Q}},N'\rangle$ and $\langle P,N'\rangle \to^* \langle \underline{\underline{P}},N\rangle$. By inspection of the semantics, we therefore have

$$\langle P \mid Q, M\rangle \rightarrow^{*} \langle P \mid \underline{\underline{Q}}, N'\rangle \rightarrow^{*} \langle \underline{\underline{P}} \mid \underline{\underline{Q}}, N\rangle$$

as required. □

Theorem 11. *If* $\forall M.(\Phi\langle Q, M\rangle$ *and* $\langle P, M\rangle \rightarrow^{*} N) \Rightarrow \Phi\langle Q, N\rangle$, *then*

$$P \circ Q \leqslant P \mid Q$$

Proof. There are two situations to consider:
1. $(M, \perp) \in \mathscr{B}(P \circ Q)$: If $P \circ Q$ may diverge on M, then either Q may diverge on M or $\langle Q, M\rangle \rightarrow^{*} N'$ and P may diverge on N'. In either case, $P \mid Q$ may diverge on M as well; thus $(M, \perp) \in \mathscr{B}(P \mid Q)$ as required.
2. $(M, N) \in \mathscr{B}(P \circ Q)$: By Lemma 9, there is an N' such that $(M, N') \in \mathscr{B}(Q)$ and $(N', N) \in \mathscr{B}(P)$. Thus, $\Phi\langle P, N\rangle$ and also $\Phi\langle Q, N\rangle$ (which follows from the assumption since $\Phi\langle Q, N'\rangle$ holds and $\langle P, N'\rangle \rightarrow^{*} N$). By the semantics, these two postconditions entail $\Phi\langle P \mid Q, N\rangle$, that is

   $$\langle \underline{\underline{P}} \mid \underline{\underline{Q}}, N\rangle \rightarrow N$$

   which, together with Lemma 10, gives the required result. □

2.4. Program laws

The program which "does nothing" – one which can never perform any reactions and therefore can only terminate – will be represented by a single reaction-action pair $(A \Leftarrow \mathit{False})$. Since the reaction condition is false, the action A, and arity are irrelevant.

Definition 12. Let Δ denote the canonical representative "skip" program, equivalent to $(A \Leftarrow \mathit{False})$ for arbitrary A.

With respect to the basic input–output partial correctness ordering $\leqslant$ it is clear that Δ obeys the usual "skip" laws of being an identity for sequential and parallel composition.

We present a collection of laws in Fig. 2. A number of laws involve residual programs (Definition 6). The residual program satisfies interesting laws because it expresses concisely the termination synchronisation requirement of the program with its context. The proof of most of these laws is straightforward. We just sketch three cases:
1. $P \circ \Delta \equiv P$
2. $P \leqslant P \mid P$
3. $(P \mid Q) \circ R \leqslant P \mid (Q \circ R)$

Proof. (1) Notice that $\mathscr{B}(\Delta) = \{(M, M) \mid M \in \mathbf{M}\}$. In the following let ? be a multiset or $\perp$. Then if $(M, ?) \in \mathscr{B}(P \circ \Delta)$, we also have $(M, ?) \in \mathscr{B}(P)$ and if $(M, ?) \in \mathscr{B}(P)$, then we also have $(M, ?) \in \mathscr{B}(P \circ \Delta)$. This gives the desired equivalence.

The Sequential Laws
1. $P \circ (Q \circ R) \equiv (P \circ Q) \circ R$
2. $P \circ \Delta \equiv P \equiv \Delta \circ P$

The Parallel Laws
1. $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$
2. $P \mid Q \equiv Q \mid P$
3. $P \equiv \Delta \mid P$
4. $P \leqslant P \mid P$

The Parallel-Sequential Laws
1. $(P \mid Q) \circ R \leqslant P \mid (Q \circ R)$
2. $(P_1 \mid P_2) \circ (Q_1 \mid Q_2) \leqslant (P_1 \circ Q_1) \mid (P_2 \circ Q_2)$
3. $P \circ (Q \mid R) \leqslant (P \circ Q) \mid (P \circ R)$

The Residual-Program Laws
1. $(P \mid \underline{R}) \circ (Q \mid R) \leqslant (P \circ Q) \mid R$
2. $(P \equiv \underline{P}) \Rightarrow (P \equiv P \mid P)$

Fig. 2. Summary of laws.

(2) If $(M, \perp) \in \mathscr{B}(P)$ then the parallel composition $P \mid P$ may also diverge. Suppose $(M, N) \in \mathscr{B}(P)$. Thus, by Proposition 7,

$$\langle P, M\rangle \rightarrow^* \langle \underline{P}, N\rangle \rightarrow N$$

Consideration of the semantics shows that steps which change the program in a configuration, do not change the multiset. Thus, there is a derivation sequence for $\langle P \mid P, M\rangle$ which changes the two instances of P in adjacent steps and

$$\langle P \mid P, M\rangle \rightarrow^* \langle \underline{P} \mid \underline{P}, N\rangle \rightarrow N$$

The result follows.

(3) Suppose that $(M, \perp) \in \mathscr{B}((P \mid Q) \circ R)$ then either $(M, \perp) \in \mathscr{B}(R)$, or $\langle R, M\rangle \rightarrow^* N'$ and $(N', \perp) \in \mathscr{B}(P \mid Q)$. In the first case, $(M, \perp) \in \mathscr{B}(Q \circ R)$ and thus $(M, \perp) \in \mathscr{B}(P \mid (Q \circ R))$. In the second case, by Proposition 7 and the semantics:

$$\langle P \mid (Q \circ R), M\rangle \rightarrow^* \langle P \mid (Q \circ \underline{R}), N'\rangle \rightarrow \langle P \mid Q, N'\rangle$$

and the result follows.

Suppose that $(M, N) \in \mathscr{B}((P \mid Q) \circ R)$. By Lemma 9, there is an N' such that $(M, N') \in \mathscr{B}(R)$ and $(N', N) \in \mathscr{B}(P \mid Q)$. By Proposition 7 and the semantics:

$$\langle P \mid (Q \circ R), M\rangle \rightarrow^* \langle P \mid (Q \circ \underline{R}), N'\rangle \rightarrow \langle P \mid Q, N'\rangle \rightarrow^* N \qquad \square$$

To understand why $P \not\equiv P \mid P$ in general, consider

P_1: $(n \rightarrow n - 1 \Leftarrow n > 0)$

P_2: $(n \rightarrow n + 1 \Leftarrow n < 1)$

P: $P_1 \circ P_2$

applied to the multiset $\{0\}$. P increments the 0, decrements it and terminates; $P \mid P$ may not terminate:

$$\begin{aligned}&\langle P \mid P, \{0\}\rangle \to \langle P \mid P, \{1\}\rangle \to\\ &\langle P_1 \mid P, \{1\}\rangle \to \langle P_1 \mid P, \{0\}\rangle \to\\ &\langle P_1 \mid P, \{1\}\rangle \to \dots\end{aligned}$$

Notice that the example makes essential use of sequential composition; the second law for residual programs shows that the expected equivalence holds when P is simple.

2.5. *Interference-freedom: stability, separability and left exclusivity*

One way in which the parallel and sequential compositions of two given programs can be related by the refinement relation is if certain "noninterference" conditions are satisfied. Here we consider some noninterference conditions which are expressible at the level of the individual elements within the multiset.

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

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

where x is some multiset element and $\bar{a}$ is a tuple of elements. Informally, $\overline{R}_M(x)$ says that if x is in any tuple of elements from M, then R is false for that tuple. 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 *separable* if the sets of elements that satisfy them are guaranteed to be disjoint:

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

An important notion is that of an element being stable with respect to some program: roughly speaking, a stable element for a given program can never influence a computation step in any multiset.

Definition 13 (*Stability*). An element, e, is stable for a program P if and only if for all multisets M:
1. $\langle P, M\rangle \to^* M' \Rightarrow \langle P, M \uplus \{e\}\rangle \to^* M' \uplus \{e\}$.
2. $\langle P, M \uplus \{e\}\rangle \to^* \langle P', M'\rangle \Rightarrow \langle P, M\rangle \to^* \langle P', M''\rangle$ where $M' = M'' \uplus \{e\}$.

As a consequence of this definition, we immediately have:

Fact 14. *If $\overline{R}(e)$ then e is stable for $(A \Leftarrow R)$.*

We will need an asymmetric notion of exclusivity between programs:

Definition 15. $(A \Leftarrow R)$ and $(A' \Leftarrow R')$ are *left exclusive*, $Lex((A \Leftarrow R),(A' \Leftarrow R'))$ if and only if $Separable(R,R')$ and

$$(\forall x_1,\dots,x_n.R(x_1,\dots,x_n))\text{implies}(\forall a \in A(x_1,\dots,x_n).\overline{R'}(a))$$

This notion lifts to programs in the following way:

$Lex(P,Q)$ if and only if $\bigwedge\{Separable(R,R') \mid R \in P, R' \in Q\} \wedge$
$\bigwedge\{(\forall x_1,\dots,x_n.R(x_1,\dots,x_n))\text{implies}(\forall a \in A(x_1,\dots,x_n).\overline{R'}(a)) \mid (A \Leftarrow R) \in P, R' \in Q\}$.

The symbol $\in$ is overloaded in Definition 15 but no confusion should arise from this abuse of notation. $R' \in Q$ means that the reaction R' appears in the text of Q (similarly for $(A \Leftarrow R) \in P$). The notation $a \in A(x_1,\dots,x_n)$ means that the value a is a member of the result of $A(x_1,\dots,x_n)$. As an illustration of this definition, we have $Lex(match, init)$, where *match* and *init* are as defined in the Introduction.

We close this section with a general result relating sequential and parallel composition; this proves useful later in the pipelining transformation:

Theorem 16. *For any programs P and Q,*

$$Lex(P,Q) \Rightarrow P \mid Q \equiv P \circ Q$$

Proof. (1) $P \circ Q \leqslant P \mid Q$: Suppose we have some M with $\Phi(Q,M)$ and $\langle P,M\rangle \rightarrow^* N$, then $\Phi(Q,N)$ follows because $Lex(P,Q)$. The result follows from Theorem 11.

(2) $P \mid Q \leqslant P \circ Q$: $Lex(P,Q)$ enforces that P and Q consume disjoint sets of elements and that the results produced by P cannot affect Q. Consider the behaviours:

(a) $(M,\perp) \in \mathscr{B}(P \mid Q)$:
Either:
- there is a multiset $M' \subseteq M$ with $(M',\perp) \in \mathscr{B}(Q)$, in which case $(M,\perp) \in \mathscr{B}(Q)$ and thus $(M,\perp) \in \mathscr{B}(P \circ Q)$, or
- $M = M' \uplus M''$, $\langle Q,M'\rangle \rightarrow^* N$ and $\langle P,M'' \uplus N\rangle\uparrow$. But then $(M,\perp) \in \mathscr{B}(P \circ Q)$.

(b) $(M,N) \in \mathscr{B}(P \mid Q)$:
Since $Lex(P,Q)$ there is a multiset N' such that $\langle Q,M\rangle \rightarrow^* N'$ and $\langle P,N'\rangle \rightarrow^* N$. Thus $(M,N) \in \mathscr{B}(P \circ Q)$.

Thus we have $\mathscr{B}(P \mid Q) \subseteq \mathscr{B}(P \circ Q)$. $\square$

3. Pipelining transformation

A natural style of programming involves the decomposition of a task into components which are then sequentially composed. This style of programming is familiar from functional programming. Understanding of the properties of the individual components enables compositional reasoning about the properties of the whole program. Unfortunately, in a parallel setting, this style fails to take any advantage of the potential for concurrent execution of the individual components. The purpose of this section is to show how a program constructed from sequential composition can be transformed into a pipeline program: one in which the sequence of tasks is connected in such a way that the output of one task feeds piecemeal into the input of the next. The outcome of this transformation is that sequential composition, $\circ$, is "replaced" by parallel combination, $|$.

3.1. An example

We start with a motivating example, based on various combinations of the following three rules:

$$P_1\colon (n \rightarrow n-1, n-2 \Leftarrow n>1)$$
$$P_2\colon (n \rightarrow 1 \Leftarrow n=0)$$
$$P_3\colon (n, m \rightarrow n+m)$$

We start by considering the straightforward sequential composition of these rules:

$$P_3 \circ P_2 \circ P_1$$

If we apply this to the multiset $\{n\}$, it will terminate with a singleton multiset containing the nth fibonnaci number. This composition effectively builds the recursion tree and then collapses it with the third rule. P_2 acts as an interface between the two processes.

Proposition 17. $P_3 \circ P_2 \circ P_1 \equiv P_3 \circ (P_2 \mid P_1)$.

Proof. It is straightforward to verify $Separable(\lambda n.n>1, \lambda n.n=0)$ and $\forall a \in (\lambda n.1)x.\ \overline{\lambda n.n>1}(a)$. Thus, Theorem 16 applies and the result follows because $\equiv$ is a congruence with respect to $\circ$. $\square$

Unfortunately, we cannot replace the remaining $\circ$ by a $|$. The reaction condition of P_3 is not exclusive with either of the other conditions; as a consequence putting P_3 in parallel with $P_2 \mid P_1$ could lead to incorrect results (by deleting 0 elements) or nontermination (because of the interaction between P_3 and P_1). The sequential composition encodes an essential producer/consumer relationship. However, a more parallel program can be produced by placing an *interface* between P_3 and $P_2 \mid P_1$ to prevent interference whilst letting data pass to P_3 in a piecemeal fashion.

We proceed by developing a general transformation scheme for pipelining before returning to this example.

3.2. The pipelining transformation

We consider a program $P_2 \circ P_1$. To enable P_2 to consume the identified stable elements of P_1 concurrently, we must ensure that P_2 does not interfere with the unstable elements. This is achieved by *tagging* the stable elements and modifying P_2 so that it can only operate on tagged data. We introduce a pair of generic encoding and decoding programs.

Definition 18. Given some tag, τ, we define the pair of encoding/decoding functions, γ_τ and δ_τ:

$$\gamma_\tau\text{: } (x \rightarrow (\tau, x) \Leftarrow \neg \mathit{ispair}(x) \vee (\mathit{fst}\ x \neq \tau))$$

$$\delta_\tau\text{: } (x \rightarrow \mathit{snd}(x) \Leftarrow \mathit{ispair}(x) \wedge (\mathit{fst}\ x = \tau))$$

where *fst*, *snd* are the first and second projections on pairs, respectively, and *ispair* is a predicate that tests if its argument is a pair.

We define S to be a multiset of τ-free elements, and T to be the multiset of τ-encoded elements, $\{(\tau, e) \mid e \in S\}$. It is easy to verify the following:

Lemma 19.

$$\delta_\tau \circ \delta_\tau \equiv_{S \cup T} \delta_\tau \qquad (1)$$

$$\gamma_\tau \circ \delta_\tau \equiv_{S \cup T} \gamma_\tau \qquad (2)$$

$$\delta_\tau \circ \gamma_\tau \equiv \delta_\tau \qquad (3)$$

$$\delta_\tau \circ \gamma_\tau \circ \delta_\tau \equiv \delta_\tau \qquad (4)$$

$$\gamma_\tau \circ \gamma_\tau \equiv \gamma_\tau \qquad (5)$$

We wish to define an encoding operation on programs, such that the encoded version operating on a tagged version of the data behaves just like the original, modulo tagging.

An encoding of P is specified by a tag and the transformation function; with respect to relational behaviour, such an encoding, (τ, Ψ) is *correct* if

$$\delta_\tau \circ P \equiv_S \delta_\tau \circ \Psi(P) \circ \gamma_\tau$$

There are a number of alternative, correct encodings. For example, $(\tau, \lambda p.\gamma_\tau \circ p \circ \delta_\tau)$ is a correct encoding for any program P. It is easy to see that if neither P, nor any of its derivatives, introduce τ-tagged elements, then correctness is equivalent to

$$P \equiv_S \delta_\tau \circ \Psi(P) \circ \gamma_\tau$$

The whole purpose of encoding is to obtain interference freedom, and so the trivial transformation given above will not be adequate. The following transformation will provide a correct encoding operation and give us the interference freedom we will need.

Definition 20. For simplicity of presentation we assume that basic programs have unary reactions and actions (the generalisation is straightforward) and that the actions produce

k elements. The transformation ψ_τ is defined inductively as follows:

$$\begin{aligned}\psi_\tau((A(x) \Leftarrow R(x))) &= (\lambda(y_1,\dots y_k).(\tau,y_1)\dots(\tau,y_k))A(snd\,x)\\ &\qquad \Leftarrow ispair(x) \wedge (fst\,x = \tau) \wedge R(snd\,x)\\ \psi_\tau(P \circ Q) &= \psi_\tau(P) \circ \psi_\tau(Q)\\ \psi_\tau(P \mid Q) &= \psi_\tau(P) \mid \psi_\tau(Q)\end{aligned}$$

Proposition 21. *For any multiset M, let $\tau \cdot M$ denote the corresponding multiset of tagged elements $\{\!|(\tau,a) \mid a \in M|\!\}$. For all P, M,*
1. $\langle P,M\rangle \rightarrow \langle P',M'\rangle \Rightarrow \langle \psi_\tau(P), \tau\cdot M\rangle \rightarrow \langle \psi_\tau(P'), \tau \cdot M'\rangle$.
2. $\langle \psi_\tau(P), \tau\cdot M\rangle \rightarrow \langle Q,N\rangle \Rightarrow \exists P',M'. \langle P,M\rangle \rightarrow \langle P',M'\rangle \wedge \psi_\tau(P') = Q \wedge \tau\cdot M' = N$.

Proof. Inductions on the structure of the proof of the respective one-step transitions. □

Proposition 22. *For all programs, P, there is a tag, τ, such that (τ,ψ_τ) is a correct transformation for P.*

Proof. We need to show that $\delta_\tau \circ \psi_\tau(P) \circ \gamma_\tau \equiv_S \delta_\tau \circ P$ where S is any multiset of τ-free elements. The precomposition of $\psi_\tau(P)$ by $\gamma_\tau \circ \delta_\tau$, and of P by δ_τ guarantees that the P obtains a multiset with no tagged elements, and $\psi_\tau(P)$ obtains the same multiset with exactly one level of tagging. Thus, we can apply the previous proposition to show that each step of one program can be simulated by the other. Finally, applying δ_τ to multisets N and $\tau \cdot N$ yields the same result. □

A useful property of a τ-encoding, which follows easily from the definition, is that unencoded elements are stable for $\psi_\tau(P)$ (see Definition 13) – i.e. they cannot partake in any reaction.

Now supposing that we have a program, *int* which detects and tags *all* stable elements for P_1, then we can implement $P_2 \circ P_1$ by $\delta_\tau \circ (\psi_\tau(P_2) \mid int \mid P_1)$ for some suitable τ. Unfortunately, it is not always possible to write an interface to detect all stable elements; instead we specify the properties that we expect an interface to satisfy. Notice that Δ[2] satisfies this specification (the "bottom" interface).

Definition 23. A program, *int* is an interface for the program P, if
1. its only action is to tag elements:

$$int\colon (x \rightarrow (\tau,x) \Leftarrow (\neg\, ispair(x) \vee (fst\ x \neq \tau)) \wedge C(x))$$

for some condition C dependent on x.

[2] Recall that $\Delta \equiv (A \Leftarrow False)$ for arbitrary action A.

2. $\delta_\tau \circ (int \,|\, P) \leqslant \delta_\tau \circ P$

Now suppose that we have such an interface for P_1 then we have the following result:

Proposition 24. *If τ is a new tag, P_2 is simple and does not produce τ-coded elements and τ-coded elements are stable for P_1:*

$$\delta_\tau \circ (\gamma_\tau \,|\, \underline{\underline{\psi_\tau(P_2)}}) \circ (\psi_\tau(P_2) \,|\, int \,|\, P_1) \leqslant_S P_2 \circ P_1$$

for any multiset of τ-free elements, S.

Proof. Since τ is new and P_2 does not produce τ-coded elements, we have $\delta_\tau \circ P_2 \circ \delta_\tau \equiv_S P_2$. Thus,

$$\begin{array}{rll}
P_2 \circ P_1 & \equiv_S \ \delta_\tau \circ P_2 \circ \delta_\tau \circ P_1 & \\
& \geqslant \ \delta_\tau \circ \psi_\tau(P_2) \circ \gamma_\tau \circ \delta_\tau \circ P_1 & \\
& \geqslant \ \delta_\tau \circ \psi_\tau(P_2) \circ \gamma_\tau \circ \delta_\tau \circ (int \,|\, P_1) & \\
& \equiv \ \delta_\tau \circ \psi_\tau(P_2) \circ \gamma_\tau \circ (int \,|\, P_1) & \\
& \geqslant \ \delta_\tau \circ (\psi_\tau(P_2) \,|\, \gamma_\tau \circ (int \,|\, P_1)) & \text{from Left-exclusivity} \\
& \geqslant \ \delta_\tau \circ (\gamma_\tau \,|\, \underline{\underline{\psi_\tau(P_2)}}) \circ (\psi_\tau(P_2) \,|\, int \,|\, P_1) & \text{by residual program laws.} \quad \square
\end{array}$$

The rightmost element of the composition causes P_1 and $\psi_\tau(P_2)$ to be executed in parallel with the interface mediating between them. The interface may not tag all stable elements (consider Δ) and thus, when the rightmost element terminates there may still be some elements which have not been processed by $\psi_\tau(P_2)$; hence the second element of the composition. We would like to simplify this result by omitting the second component; we can do this when the two components have the same postcondition, as shown in the following:

Proposition 25. *If $\Phi(\psi_\tau(P_2) \,|\, int \,|\, P_1, M) \Rightarrow \Phi(\gamma_\tau \,|\, \underline{\underline{\psi_\tau(P_2)}}, M)$ then:*

$$\delta_\tau \circ (\psi_\tau(P_2) \,|\, int \,|\, P_1) \leqslant \delta_\tau \circ (\gamma_\tau \,|\, \underline{\underline{\psi_\tau(P_2)}}) \circ (\psi_\tau(P_2) \,|\, int \,|\, P_1)$$

Proof. We consider two cases:

1. For all M

$$\begin{array}{rl}
\langle \delta_\tau \circ (\psi_\tau(P_2) \,|\, int \,|\, P_1), M \rangle \uparrow & \Rightarrow \ \langle (\psi_\tau(P_2) \,|\, int \,|\, P_1), M \rangle \uparrow \\
& \Rightarrow \ \langle \delta_\tau \circ (\gamma_\tau \,|\, \underline{\underline{\psi_\tau(P_2)}}) \circ (\psi_\tau(P_2) \,|\, int \,|\, P_1), M \rangle \uparrow
\end{array}$$

2. Suppose $\langle \delta_\tau \circ (\psi_\tau(P_2) \,|\, int \,|\, P_1), M \rangle \longrightarrow^* N$ for some N. Then there is some multiset N' such that

$$\langle \psi_\tau(P_2) \,|\, int \,|\, P_1, M \rangle \longrightarrow^* N' \quad \text{and} \quad \Phi(\psi_\tau(P_2) \,|\, int \,|\, P_1, N')$$

But then, by assumption, $\Phi(\gamma_\tau \mid \underline{\underline{\psi_\tau(P_2)}}, N')$ and thus:

$$\langle \delta_\tau \circ (\gamma_\tau \mid \underline{\underline{\psi_\tau(P_2)}}) \circ (\psi_\tau(P_2) \mid int \mid P_1), M\rangle \to^* N \qquad \square$$

In particular, it is easy to verify that this proposition does apply if *int* is complete (i.e. tags all stable elements).

3.3. The example revisited

Returning to our example the only stable elements for $P_2 \mid P_1$ are the 1s; since it is possible to define a complete interface, the program $P_3 \circ (P_2 \mid P_1)$ can be transformed to

$$\delta_\tau \circ (\psi_\tau(P_3) \mid int \mid P_2 \mid P_1)$$

where

$$int\colon (x \to (\tau, 1) \Leftarrow x = 1)$$

To close this section we present another example of this pipelining transformation. We consider the prime factorisation problem, presented in [5], which utilises the result from number theory that any number can be written uniquely as a product of primes. The program has a number as its input and produces a multiset of primes, each prime factor of the input being repeated the number of times that it is used in the factorisation.

We define the initial program as follows:

$$factor(n)\colon (P_6 \circ P_5 \circ P_4 \circ P_3 \circ P_2 \circ P_1)\{(n,n)\}$$

where

P_1: $((a,b) \to (a,b,0),(a-1,b) \Leftarrow a \geqslant 3)$

P_2: $((a,b) \to (a,b,0) \Leftarrow a < 3)$

P_3: $(((x,a,b),(y,c,d)) \to (y,c,d) \Leftarrow multiple(x,y))$

P_4: $((n_1,n_2,k) \to (n_1,n_2/n_1,k+1) \Leftarrow multiple(n_2,n_1))$

P_5: $((n_1,n_2,k) \to (n_1,n_2,k-1),n_1 \Leftarrow k \geqslant 1)$

P_6: $((n_1,n_2,k) \to \; \Leftarrow k = 0)$

where we have used pattern matching on the bound variables in order to avoid complicating the reaction conditions.

P_1 and P_2 together produce a (multi)set of triples such that each element consists of a number less than or equal to the original input, the original input and 0; P_3 removes all of the triples which do not have a prime number as their first component; P_4 increments the third component of the triples to record the number of times that the prime first element divides the input; P_5 generates copies of the primes; and P_6 deletes redundant triples. We make the following observations about factor:

$$P_3 \circ P_2 \circ P_1 \equiv P_3 \mid P_2 \mid P_1$$

and

$$P_6 \circ P_5 \equiv P_6 \mid P_5$$

which both follow from Theorem 16.

We can define an interface, *int*, for $P_4 \circ (P_3 \mid P_2 \mid P_1)$:

$$\mathit{int}\colon ((n_1, n_2, k) \to (\tau, (n_1, n_2, k)) \Leftarrow \neg \mathit{multiple}(n_2, n_1))$$

It is routine to verify that

$$\delta_\tau \circ (\mathit{int} \mid (P_4 \circ (P_3 \mid P_2 \mid P_1))) \leqslant (P_4 \circ (P_3 \mid P_2 \mid P_1))$$

and

$$\Phi(\psi_\tau(P_6) \mid \psi_\tau(P_5) \mid \mathit{int} \mid (P_4 \circ (P_3 \mid P_2 \mid P_1)), M) \Rightarrow \Phi(\gamma_\tau \mid \underline{\underline{\psi_\tau(P_6)}} \mid \underline{\underline{\psi_\tau(P_5)}}, M)$$

and consequently, *factor*(n) can be implemented by

$$\delta_\tau((\psi_\tau(P_6) \mid \psi_\tau(P_5) \mid \mathit{int} \mid (P_4 \circ (P_3 \mid P_2 \mid P_1)))\{(n, n)\})$$

A final optimisation which is possible is to omit the tagging of the second component of the right hand side of $\psi_\tau(P_5)$; such single elements can play no further part in the computation.

4. Conclusion

In this paper, we have introduced a notation for composing parallel programs which has been used to enhance the original Gamma language. In order to put this work into perspective, we first provide a sketch of formalisms akin to Gamma and alternative views of program composition in Gamma. Then, we summarise ongoing work on Gamma and suggest avenues for further research.

4.1. Related work

Some languages bearing similarities with the chemical reaction paradigm have been proposed in the literature. Let us briefly review the most significant ones:

- A Unity program [11] is basically a set of multiple-assignment statements. Program execution consists in selecting nondeterministically (but following a fairness condition) some assignment statement, executing it and repeating forever. [11] defines a temporal logic for the language and the associated proof system is used for the systematic development of parallel programs. Some Unity programs look very much

like Gamma programs (an example is the exchange sort program presented in the introduction). The main departures from Gamma is the use of the array as the basic data structure and the absence of locality property. On the other hand, Unity allows the programmer to distinguish between synchronous and asynchronous computations which makes it more suitable as an effective programming language for parallel machines.
- In the same vein as Unity, the *action systems* presented in [4] are *do-od* programs consisting of a collection of guarded atomic actions, which are executed nondeterministicly so long as some guard remains true.
- Linda [17, 9] contains a few simple commands operating on a tuple space. A producer can add a value to the tuple space; a consumer can read (destructively or not) a value from the tuple space. Linda is a very elegant communication model which can easily be incorporated into existing programming languages.
- LO [3] (for Linear Objects) was originally proposed as an integration of logic programming and object-oriented programming. It can be seen as an extension of Prolog with formulae having multiple heads. From an object-oriented point of view, such formulae are used to implement methods. A method can be selected if its head matches the goal corresponding to the object in its current state. The head of a formula can also be seen as the set of resources consumed by the application of the method (and the tail is the set of resources produced by the method). LO has been used as a foundation for *interaction abstract machines*, extending the chemical reaction metaphor with a notion of broadcast communication: subsolutions (or "agents") can be created dynamically and reactions can have the extra effect of broadcasting a value to all the agents.

A different approach to the introduction of composition operators in Gamma is taken in [13]. Their solution is based on a separation of reduction rules into proper transformations (which correspond to individual chemical reactions) and unproper transformations which modify the program but have no effect on the multiset. The resulting definition of the parallel operator restricts its non determinism and makes it possible to avoid some undesired computations. Two observational equivalences based on the concept of bisimulation are defined and are shown to be congruences and they are characterised by means of sound and complete axiomatisations [13].

Several proposals have been made recently for enhancing Gamma with better facilities for expressing control. Let us mention in particular higher-order versions of Gamma [19, 14] which make it possible to manipulate reactions just as ordinary data (allowing the programmer to define his own composition operators) and the language of *schedules* [12]. The idea behind schedules is to separate the definition of a Gamma program in two parts: individual reactions, which correspond to a single application of a rewrite rule, and schedules, which specify the control part of the program. The language of schedules includes iteration, sequential and parallel composition, non-determinism. A nice property of schedules is that they disentangle the two orthogonal features of Gamma (the choice of multisets as the data structure and the "stirring mechanism" as the control structure) and they allow the user to make his own choice concerning the control component of the program. A notion of refinement is also defined in [12] in terms of degree of determinism of schedules.

The interested reader can find in [6] a more comprehensive account of the chemical reaction paradigm including various examples, a discipline of programming based on a set of program schemes called *tropes*, different extensions and implementation issues.

4.2. *Ongoing work and perspectives*

4.2.1. *Structured Gamma*

The choice of the multiset as the unique data constructor may lead to programs which are unnecessary complex when the programmer needs to encode specific data structures. For example, it is necessary to resort to pairs (*index*, *value*) to represent sequences in a sort program. Trees or graphs can be encoded in a similar way. This lack of structuring is detrimental both for reasoning about programs and for implementing them. It is important to circumvent this problem without jeopardising the basic qualities of the language. Let us point out in particular that it would not be acceptable to take the usual view of recursive type definitions because this would lead to a recursive style of programming and ruin the fundamental locality principle (because the data structure would then be manipulated as a whole).

To solve this problem, we have proposed an enhancement of Gamma based on a notion of *structured multiset*. A structured multiset can be seen as a set of addresses satisfying specific relations and associated with a value [16]. A type is defined in terms of rewrite rules and a structured multiset belongs to a type T if its underlying set of addresses satisfies the invariant expressed by the rewrite system defining T. In contrast with the local conditions used in this paper, structured multisets can be seen as global properties of the multiset.

A reaction in Structured Gamma can:
- Test and modify the relations on addresses.
- Test and modify the values associated with the addresses.

The significance of the approach is that the programmer can define his own types and programs can be checked according to the type definitions. This verification can be made automatically using term rewriting techniques [16].

A promising application of this idea concerns the definition and analysis of software architectures: in this context, values are the individual entities to be coordinated (agents or processes) and the relations represent their communication capabilities. The invariant is a property of the communication structure (e.g. ring, star, etc.). The interested reader can find more information on this application in [20].

A natural avenue for further research is the extension of the results presented in this paper to Structured Gamma. The extra information provided by the structured types can be useful to allow further transformations. We are also studying the relevance of the ordering introduced in Section 2 for software architectures. A notion of refinement is crucial in this context (to decide when an architecture is a correct implementation of another, more general one). There does not seem to be a single answer to this problem because different usages may put different requirements on the notion of refinement. For instance, security-related properties may be preserved through refinements corresponding to multiset inclusion (because removing links or entities decrease the global information flow), but this form of refinement may not be acceptable for functional properties (because removing links or entities may alter the services provided by the system).

4.2.2. *Compositional semantics*

As mentioned earlier, the behavioural equivalence ($\equiv$) used in this paper is not a congruence with respect to $|$. Because of the lack of a general substitutivity property, the use of the relational ordering in reasoning about programs is limited. Nevertheless, there is rich selection of refinement laws which do indeed respect the parallel composition operator. These are studied in [23, 22]; in these earlier papers, only partial correctness was studied but the latter paper also considered different composition operators.

The equivalence used in this paper is based on input–output behaviours of programs. A first idea to get a more precise notion of equivalence would be to define it in terms of intermediate states rather than just input–output. But it is well-known from the study of state-based concurrency that it is insufficient to use sequences of states as a means of distinguishing programs. The reason why a semantics based on state-sequences still does not yield a compositional definition is that it does not take into account the possible interference (from the program's surrounding context) that can occur during execution. In order to solve this problem, we can think of the computational model as a form of shared-variable language, and seek inspiration from techniques developed in that context.

The papers [23, 22] adapt a standard approach used in the semantics of shared-variable concurrency based on sequences of multiset pairs: [1, 21, 15] and in particular Brookes' variant [8]. The idea is to define the meaning of a program P as a set of nonempty finite sequences of multiset pairs. The transition traces describing the finite (terminating) behaviours of a program, specified as in [8], is given by the set

$$\begin{aligned}
&\{(M_0,N_0)(M_1,N_1)\ldots(M_k,N_k) \mid \\
&\quad \langle P,M_0\rangle \rightarrow^* \langle P_1,N_0\rangle \wedge \\
&\quad \langle P_1,M_1\rangle \rightarrow^* \langle P_2,N_1\rangle \wedge \cdots \wedge \langle P_k,M_k\rangle \rightarrow^* N_k \ \}
\end{aligned}$$

The intuition behind the use of transition traces is that each transition trace

$$(M_0,N_0)(M_1,N_1)\ldots(M_k,N_k) \in \mathsf{T}[\![P]\!]$$

represents a terminating execution of program P in some context, starting with multiset M_0, and in which each of the pairs (M_i,N_i) represents computation steps performed by (derivatives of) P and the adjacent multisets N_{i-1},M_i represent possible interfering computation steps performed by the "context".

A key feature of Brookes' definition of transition traces is that they use the reflexive-transitive closure of the one-step evaluation relation, $\rightarrow^*$. A consequence of this reflexivity and transitivity is that they are closed under "stuttering" and "absorption" properties described below. In the following let ε denote the empty sequence.

Definition 26. Let α range over finite sequences of multiset pairs, and β range over finite or infinite sequences. A set $T \subseteq \wp((\mathbf{M}\times\mathbf{M})^+) \cup \wp((\mathbf{M}\times\mathbf{M})^\omega)$[3] is closed under left-stuttering[4] and absorption if it satisfies the following two conditions

$$\textbf{left-stuttering}\ \frac{\alpha\beta\in T,\ \beta\neq\varepsilon}{\alpha(M,M)\beta\in T} \qquad \textbf{absorption}\ \frac{\alpha(M,N)(N,M')\beta\in T}{\alpha(M,M')\beta\in T}$$

Let $\ddagger T$ denote the left-stuttering and absorption closure (henceforth just closure) of a set T.

It is easy to see that the finite transition traces of a program are closed, and can be obtained from the traces of atomic steps ($\rightarrow$) by closure. We extend the definition to all the transition traces of the program using this closure operation on the *atomic* traces:

Definition 27. The *atomic* traces of a program, $\mathsf{T}[\![P]\!]$, are the finite and infinite sequences of pairs of multisets, given by

$$\begin{aligned}
\mathsf{T}[\![P]\!] = {} & \{(M_0,N_0)(M_1,N_1)\dots(M_k,N_k) \mid \\
& \langle P,M_0\rangle \rightarrow \langle P_1,N_0\rangle \wedge \\
& \langle P_1,M_1\rangle \rightarrow \langle P_2,N_1\rangle \wedge\cdots\wedge \langle P_k,M_k\rangle \rightarrow N_k\} \\
& \cup \\
& \{(M_0,N_0)(M_1,N_1)\dots(M_i,N_i)\dots \mid \\
& \langle P,M_0\rangle \rightarrow \langle P_1,N_0\rangle \wedge \langle P_i,M_i\rangle \rightarrow \langle P_{i+1},N_i\rangle, i\geqslant 1\}
\end{aligned}$$

The *transition traces* of a program are given by the closure of the atomic traces: $\ddagger\mathsf{T}[\![P]\!]$.

Sands [23, 22] present a compositional definition of the transition trace semantics, introduce an ordering based on traces and verify a number of compositional program laws. All of the laws presented earlier in this paper can be verified for the new ordering using the transition trace semantics; many of the proofs are more straightforward. However, Proposition 7 is not valid in the compositional semantics. Since the development of the pipelining transformation makes use of this result, we cannot use the

[3] If S is a set, then S^+ will denote nonempty finite sequences of elements from S, and S^ω the infinite sequences.

[4] Notice that we say *left*-stuttering to reflect that the context is not permitted to change the state after the termination of the program. In this way each transition trace of a program only charts interactions with its context up to the point of the program's termination.

compositional semantics to validate that transformation. Interesting questions relating to the use of the compositional laws and whether the transition trace semantics is fully abstract merit further investigation.

Acknowledgements

The authors were partially funded by ESPRIT project 9102 (Coordination). They wish to thank their partners on this project, who have provided useful comments on earlier drafts of the paper. Michel Chaudron also provided valuable comments.

References

[1] K. Abrahamson, Modal logic of concurrent nondeterministic programs, in: Proc. Internat. Symp. on Semantics of Concurrent Computation, vol. 70, SV, 1979, pp. 21–33.
[2] R. Allen, D. Garlan, Formalising architectural connection, in: Proc. IEEE 16th Internat. Conf. on Software Engineering, 1994, pp. 71–80.
[3] J.-M. Andreoli, R. Pareschi, Linear objects: logical processes with built-in inheritence, New Generation Comput. 9 (1991) 445–473.
[4] R. Back, Refinement calculus, Part 2: parallel and reactive programs, in: J.W. de Bakker, W.P. de Roever, G. Rozenberg (Eds.), Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness, Lecture Notes in Computer Science, vol. 430, Springer, Berlin, 1990.
[5] J.-P. Banâtre, D. Le Métayer, The Gamma model and its discipline of programming, Sci. Comput. Programming 15 (1990) 55–77.
[6] J.-P. Banâtre, D. Le Métayer, Programming by multiset transformation, Comm. ACM 36(1) (1993) 98–111.
[7] J.-P. Banâtre, D. Le Métayer, Gamma and the chemical reaction model, in: J.-M. Andreoli, C. Hankin, D. Le Métayer (Eds.), Coordination Programming: Mechanisms, Models and Semantics, IC Press, London, 1996.
[8] S. Brookes, Full abstraction for a shared variable parallel language, in: Logic in Computer Science, IEEE Press, New York, 1993.
[9] N. Carriero, D. Gelernter, Linda in context, Comm. ACM 32(4) (1989) 444–458.
[10] N. Carriero, D. Gelernter, How to Write Parallel Programs, MIT Press, Cambridge, MA, 1990.
[11] K.M. Chandy, J. Misra, Parallel Program Design: A Foundation, Addison-Wesley, Reading, MA, 1988.
[12] M. Chaudron, E. de Jong, Towards a compositional method for coordinating Gamma programs, in: P. Ciancarini, C. Hankin (Eds.), Coordination'96 Conf., Lecture Notes in Computer Science, vol. 1061, Springer, Berlin, 1996, pp. 107–123.
[13] P. Ciancarini, R. Gorrieri, G. Zavattaro, An alternative semantics for the calculus of gamma programs, in: J.-M. Andreoli, C. Hankin, D. Le Métayer (Eds.), Coordination Programming: Mechanisms, Models and Semantics. IC Press, London, 1996.
[14] D. Cohen, J. Muylaert-Filho, Introducing a calculus for higher-order multiset programming, in: P. Ciancarini, C. Hankin (Eds.), Coordination'96 Conf., Lecture Notes in Computer Science, vol. 1061, Springer, Berlin, pp. 124–141.
[15] F.S. de Boer, J.N. Kok, C. Palamidessi, J.J.M.M. Rutten, The failure of failures in a paradigm for asynchronous communication, in: J.C.M. Baeten, J.F. Groote (Eds.), Concur'91, Lecture Notes in Computer Science, vol. 527, Springer, Berlin, 1991, pp. 111–126.
[16] P. Fradet, D. Le Métayer, Structured gamma, Tech. Report 989, IRISA, 1996.
[17] D. Gelernter, Generative communication in Linda, ACM Trans. Programming Languages Systems, 7(1) (1985) 80–112.
[18] C. Hankin, D. Le Métayer, D. Sands, A calculus of gamma programs, in: U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), Proc. 5th Workshop on Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science, vol. 757, Springer, Berlin, 1992.

[19] D. Le Métayer, Higher-order multiset programming, in: Proc. DIMACS Workshop on Specifications of Parallel Algorithms, American Mathematical Society, Providence, RI, 1994.

[20] D. Le Métayer, Software architecture styles as graph grammars, in: Proc. ACM SIGSOFT'96 4th Symp. on the Foundations of Software Engineering, ACM Press, New York, 1996.

[21] D. Park, On the semantics of fair parallelism, in: Abstract Software Specifications (1979 Copenhagen Winter School Proc.), Lecture Notes in Computer Science, vol. 86, Springer, Berlin, 1979, pp. 504–526.

[22] D. Sands, A compositional semantics of combining forms for Gamma programs, in: D. Bjoerner, M. Broy, I. Pottosin (Eds.), Formal Methods in Programming and Their Applications, Internat. Conf. Academgorodok, Novosibirsk, Russia, June/July 1993., Lecture Notes in Computer Science, vol. 735, Springer, Berlin, 1993, pp. 43–56.

[23] D. Sands, Laws of parallel synchronised termination, in: G.L. Burn, S.J. Gay, M.D. Ryan (Eds.), Theory and Formal Methods 1993: Proc. 1st Imperial College, Department of Computing, Workshop on Theory and Formal Methods, Isle of Thorns, UK, 1993; Springer, Workshops in Computer Science.