A Compositional Semantics of Combining Forms for Gamma Programs*

David SANDS [†]
Department of Computer Science,
University of Copenhagen, Universitetsparken 1,
DK-2100 Copenhagen Ø, DENMARK.
(e-mail: `dave@diku.dk`)

Abstract

The Gamma model is a minimal programming language based on local multiset rewriting (with an elegant chemical reaction metaphor); Hankin *et al* derived a calculus of Gamma programs built from basic reactions and two composition operators, and applied it to the study of relationships between parallel and sequential program composition, and related program transformations. The main shortcoming of the "calculus of Gamma programs" is that the refinement and equivalence laws described are not compositional, so that a refinement of a sub-program does not necessarily imply a refinement of the program.

In this paper we address this problem by defining a compositional (denotational) semantics for Gamma, based on the *transition trace* method of Brookes, and by showing how this can be used to verify substitutive refinement laws, potentially widening the applicability and scalability of program transformations previously described.

The compositional semantics is also useful in the study of relationships between alternative combining forms at a deeper semantic level. We consider the semantics and properties of a number of new combining forms for the Gamma model.

1 Background

The Gamma Model The Gamma formalism was proposed by Banâtre and Le Métayer [BM93] as a means for the high level description of parallel programs with a minimum of explicit control. Gamma is a minimal language based on local rewriting of a finite multiset (or *bag*), with an appealing analogy with the chemical reaction process. As an example, a program that sorts an array $a_0, \ldots, a_n$ (of integers, say) could be defined as

* *To Appear: Proceedings of the International Conference on Formal Methods in Programming and Their Applications, Novosibirsk 1993. Springer-Verlag LNCS.*

[†]Research supported by the Danish Research Council, DART Project (5.21.08.03)

follows. Represent the array as a multiset of pairs $\{(0, a_0), \ldots, (n, a_n)\}$, then just specify a single rule: "exchange ill-ordered values"

$$sort_A : ((i,x),(j,y) \to (i,y),(j,x) \Leftarrow i < j \;\&\; x > y).$$

$i < j \;\&\; x > y$ specifies a property (a *reaction condition*) to be satisfied by the selected elements (i,x) and (j,y); these elements are replaced in the multiset (the *chemical solution*) by the elements $(i,y),(j,x)$ (the product of the reaction). Nothing is said in this definition about the order of evaluation of the comparisons; if several disjoint pairs of elements satisfy the reaction condition the comparisons and replacements can even be performed in parallel.

The computation terminates when a stable state is reached, that is to say when no elements of the multiset satisfy the reaction condition (or in general, any of a number of reaction conditions). The interested reader may find a long series of examples illustrating the Gamma style of programming in [BM93]. The benefit of using Gamma in systematic program construction in the Dijkstra-Gries style are illustrated in [BM90].

A Calculus of Gamma Programs 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 composed programs. Hankin, Le Métayer and Sands [HMS92] defined two composition operators for the construction of Gamma programs from basic reactions, namely sequential composition $P_1 \circ P_2$ and parallel composition $P_1 + P_2$. 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 $P_1 + P_2$ 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—in the spirit of the original Gamma programs.

As a further simple example, consider the following program which sorts a multiset of integers:

$$\begin{array}{ll} sort_B : match \circ init & \\ \textsf{where} \quad init: & (x \to (0,x) \Leftarrow integer(x)) \\ \qquad match: & ((i,x),(i,y) \to (i,x),(i+1,y) \Leftarrow x \leq y) \end{array}$$

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.

In fact we get a sort program for a multiset of integers by placing $sort_A$ in parallel with $sort_B$ as $sort_A + sort_B$.

Hankin *et al.* [HMS92] derived a number of program refinement and equivalence laws for parallel and sequential composition, by considering the input-output behaviour induced by an operational semantics. So, for example, the program $sort_A + sort_B$ which is by definition $sort_A + (match \circ init)$ is refined by the program

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

This refinement is an instance of a general refinement law:

$$P + (Q \circ R) \geq (P + Q) \circ R.$$

Amongst other things, Hankin *et. al.* went on to describe how the laws were used in the design of a "pipelining" program transformation method.

Overview

The main shortcoming of the "calculus of Gamma programs" described in [HMS92] is that the refinement and equivalence laws described are not compositional, so that a refinement of a sub-program does not necessarily imply a refinement of the program. This limits the applicability and scalability of program transformations and inhibits the study of the relationships between alternative combining forms.

In this paper we directly address this problem by defining a compositional (denotational) semantics for Gamma, show that this can be used to verify substitutive refinement laws, and study properties of the language and relationships between alternative combining forms at a deeper semantic level.

The remainder of the paper is organised as follows. In **Section 2** we define the operational semantics of Gamma programs. In **Section 3** we describe the simple partial correctness refinement ordering and show that it is not substitutive. We then define *operational approximation* as the largest substitutive preordering contained in this.

In **Section 4** we motivate and adapt a method of Brookes [Bro93] for giving a (fully abstract) denotational semantics for a parallel shared variable *while* language originally studied by Hennessy and Plotkin [HP79]. The key technique is to give the meaning of a program as a set of *transition traces* which represent both the computation steps that the program can perform, and the ways that the program can interact with its environment.

The induced ordering from the compositional model implies operational approximation, and so in **Section 5** it is shown that a number of refinement laws can be proved. We then show that the semantics is not fully abstract (it does not completely characterise operational approximation) by proving an interesting property of the language: there is no analogy of the "Big Bang" for Gamma programs. In other words, there is no program which can add some elements to the empty set and then terminate.

Section 6 shows how the approach is useful for studying alternative combining forms for Gamma programs and the relationships between them, and **Section 7** concludes.

2 Operational Semantics

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, $P_1 + P_2$ as introduced in [HMS92].

$$P \in \mathbf{P} ::= (A \Leftarrow R) \mid P \circ P \mid P + P$$

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 $\langle P, M\rangle$, where $M \in \mathbf{M}$ is the set of finite multisets of elements. The domain of the elements is left unspecified, but is expected to include integers, booleans and tuples.

We define the single step transitions first for the individual reactions $(A \Leftarrow R)$. A reaction terminates on some multiset exactly when there is no sub-multiset of elements which (when viewed as a tuple) satisfy the reaction condition. We will not specify the details of the reaction conditions and actions, but assume, as previously, that they are total functions over tuples of integers, truth values, tuples etc.

$$\langle (A \Leftarrow R), M\rangle \to M \quad \text{if } \neg\exists \vec{a} \subseteq M.\, R\vec{a}$$

Otherwise, if we can form a tuple from elements of the multiset, and this tuple satisfies the reaction condition, the selected elements can be replaced by the elements produced by the associated action function:

$$\langle (A \Leftarrow R), M \uplus \vec{a}\rangle \to \langle (A \Leftarrow R), M \uplus A\vec{a}\rangle \quad \text{if } R\vec{a}$$

The *terminal transitions* have the form $\langle P, M\rangle \to M$, and are only defined for programs not containing sequential composition, so the remaining *terminal* transitions are defined by the following synchronised-termination rule:

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

The remaining intermediate transitions are given by first defining what we will call *active contexts.*

DEFINITION **2.1** *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 transitions are defined by the following two rules:

$$\frac{\langle P, M\rangle \to \langle P', M'\rangle}{\langle \mathbf{A}[P], M\rangle \to \langle \mathbf{A}[P'], M'\rangle} \qquad \frac{\langle Q, M\rangle \to M}{\langle \mathbf{A}[P \circ Q], M\rangle \to \langle \mathbf{A}[P], M\rangle}$$

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.

Notice that since a program does not uniquely factor into a sub-program in an active context, so this is another source of nondeterminism. We make the assumption that the one step evaluation relation is total, ie. that for all nonterminal configurations $\langle P, M\rangle$ there is at least one configuration U such that $\langle P, M\rangle \rightarrow U$. This amounts to an assumption that the transition relation on single $(A \Leftarrow R)$ pairs is total.

In [HMS92] an equivalent definition is given in terms of the more traditional SOS style, and active contexts are given as a derived construction to facilitate proofs of certain properties.

3 Operational Program Orderings

In [HMS92] defined a variety of orderings on programs based on their termination and nontermination behaviour. In this study we only consider the simplest of these, the partial correctness ordering (the $\leq_L$ order of [HMS92]).

> We define $P_1 \leq P_2$ whenever, for each possible input M, if P_1 can terminate producing some multiset N then so can P_2.

This partial correctness preorder and associated equivalence $\sim$, satisfy the following properties (a subset of those given in [HMS92], where they are stated for a stronger ordering which also treats nontermination as a possible "result"). The reaction $(A \Leftarrow \mathit{False})$ stands for a reaction whose reaction-condition is not satisfiable, hence the action A and its arity are irrelevant.

1. $P \circ (Q \circ R) \sim (P \circ Q) \circ R$
2. $P + (Q + R) \sim (P + Q) + R$
3. $P + Q \sim Q + P$
4. $(A \Leftarrow \mathit{False}) + P \sim P + (A \Leftarrow \mathit{False}) \sim P$
5. $(A \Leftarrow \mathit{False}) \circ P \sim P \circ (A \Leftarrow \mathit{False}) \sim P$
6. $P \leq P + P$
7. $(P_1 + P_3) \circ P_2 \leq (P_1 \circ P_2) + P_3$

Furthermore, the sequential composition operator is monotonic in both operands, ie. if $P_i \leq Q_i$ then $P_1 \circ P_2 \leq Q_1 \circ Q_2$. Unfortunately it is not monotonic with respect to parallel composition[1] as the following example shows:

Consider the following two atomic programs:

$$\begin{array}{l} P_1 : \ x \rightarrow 0 \Leftarrow x = 1 \\ P_2 : \ x \rightarrow 1 \Leftarrow x = 0 \end{array}$$

[1]As was also observed for the stronger "relational" ordering in [HMS92].

By property 5 we know that $(A \Leftarrow \mathit{False}) \circ P_1 \leq P_1$. Now $\langle P_1 + P_2, \{1\}\rangle$ can never terminate, but $\langle((A \Leftarrow \mathit{False}) \circ P_1) + P_2, \{1\}\rangle \rightarrow^* \{1\}$, and so $((A \Leftarrow \mathit{False}) \circ P_1) + P_2 \not\leq P_1 + P_2$.

Because of the lack of a general substitutivity property, the use of the partial correctness ordering in reasoning about programs is limited. As is standard, we are therefore interested in the largest contextual (pre) congruence relation contained in $\leq$.

Definition 3.1 *Let* **C** *range over general program contexts, then we define* operational approximation *($\sqsubseteq_o$) and* operational congruence*($\equiv_o$) respectively by:*

$$\begin{array}{lcl} P_1 \sqsubseteq_o P_2 & \iff & \forall \mathbf{C}.\, \mathbf{C}[P_1] \leq \mathbf{C}[P_2] \\ P_1 \equiv_o P_2 & \iff & P_1 \sqsubseteq_o P_2 \ \&\ P_2 \sqsubseteq_o P_1 \end{array}$$

The task now is to find some characterisation of operational approximation.

4 Transition Traces

In the previous section we showed that the partial correctness preorder was not substitutive with respect to parallel composition (and hence arbitrary program contexts). In other words, to characterise the behaviour of programs in all possible program contexts it is not enough to distinguish between them on the basis of their input-output behaviour alone.

Traces It is clear that we must consider intermediate behaviour of programs to obtain a substitutive equivalence. Consider the *strict traces*, $\mathcal{ST}[\![P]\!]$ of a program defined as all sequences of multisets which describe the single steps leading to a terminal configuration:

$$\mathcal{ST}[\![P]\!] = \{M_1, \ldots, M_k \mid \langle P, M_1\rangle \rightarrow \langle P_2, M_2\rangle \rightarrow \cdots \rightarrow \langle P_{k-1}, M_{k-1}\rangle \rightarrow M_k\}$$

Now clearly if $\mathcal{ST}[\![P_1]\!] \subseteq \mathcal{ST}[\![P_2]\!]$ then $P_1 \leq P_2$. As a possible account of program meaning, the strict traces are in some cases rather strong since they unnecessarily distinguish between $(A \Leftarrow \mathit{False})$ and $(A \Leftarrow \mathit{False}) \circ (A \Leftarrow \mathit{False})$. However, in other respects they are still not strong enough to provide a denotation for programs which is compositional, as the following example shows:

Consider the following three atomic programs:

$$\begin{array}{l} P_1 : x \rightarrow 0 \Leftarrow x = 1 \\ P_2 : x \rightarrow 2 \Leftarrow x = 1 \\ P_3 : x \rightarrow 1 \Leftarrow x = 2 \end{array}$$

Since P_1 and P_2 have the same reaction condition, it should not be too difficult to see that

$$\mathcal{ST}[\![P_1 \circ P_2]\!] = \mathcal{ST}[\![(A \Leftarrow \mathit{False}) \circ P_2]\!],$$

but that in the context $[\,] + P_3$ they give different traces; in particular,

$$\begin{array}{l}\{1\}\,\{2\}\,\{2\}\,\{1\}\,\{0\}\,\{0\} \\ \in \mathcal{ST}[\![(P_1 \circ P_2) + P_3]\!] \setminus \mathcal{ST}[\![((A \Leftarrow \mathit{False}) \circ P_2) + P_3]\!].\end{array}$$

The reason why a semantics based on traces as opposed to just input-output behaviour is still not compositional is that they do not take into account the possible interference (from the program's surrounding context) that can occur during execution. We can think of the computational model as a form of shared-variable language, and adapt standard techniques developed in that context as a possible solution.

Resumptions Hennessy and Plotkin [HP79] described a denotational semantics for a simple *while*-language with a parallel composition operator. The basis of their method is to define the meaning of a command as a *resumption*.

A resumption models the meaning of a command as a function from the current state (the store, or in our case the multiset) to a set containing all possible terminal states reachable in one step, together with a set of state-resumption pairs, each of which represents a possible next nonterminal state together with a resumption for the rest of the program. This construction can model the fact that after each atomic step in the computation of the command it may be interrupted by the context in which it is being executed. Since this interruption may change the state, the resumption component of the pair describes how the command may then resume execution.

Move Traces At about the same time, Abrahamson [Abr79], in a study of a modal logic for parallel programs, sketched a semantics for a parallel shared-state language based on sequences of "moves". A move is a pair of states representing an atomic computation step of the program. Adjacent moves model a possible interference by some other process executing in parallel with the program. A set of possible move-sequences of a program can be thought of as an "unraveling" of the program's resumption semantics, but with the advantage of being mathematically simpler to work with, since it does not involve any powerdomain constructions.

Transition Traces Brookes [Bro93] independently came up with the idea of using sequences of state-pairs to give a compositional semantics for the Hennessy-Plotkin *while*-language, but with an important improvement: by considering only those traces with certain closure properties (most importantly an *absorption* property which we will describe below) he was able to give a fully abstract semantics for the *while* language without having to add the rather unnatural co-routine operator of [HP79].

We adapt Brooke's transition trace model for our language. We define the meaning of a command as a set of transition traces, and show that the definition can be given compositionally.

4.1 Transition Trace Semantics

The idea is to define the meaning of a program P as a set of nonempty finite sequences of multiset pairs.

Definition **4.1** *The transition trace function* $\mathcal{TT}[\![\_]\!] : \mathbf{P} \to \wp((\mathbf{M} \times \mathbf{M})^{+})$ *is given by*

$$\begin{aligned}\mathcal{TT}[\![P]\!] = \ & \{(M_0, N_0)(M_1, N_1) \ldots (M_k, N_k) | \\ & \langle P, M_0\rangle \to^{*} \langle P_1, N_0\rangle \ \& \\ & \langle P_1, M_1\rangle \to^{*} \langle P_2, N_1\rangle \ \& \ldots \& \ \langle P_k, M_k\rangle \to^{*} N_k\}\end{aligned}$$

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

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

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. The partial correctness ordering is derivable from the transition traces by considering traces of length one:

Proposition **4.2**

$$P_1 \leq P_2 \iff \{(M, N) \mid (M, N) \in \mathcal{TT}[\![P_1]\!]\} \subseteq \{(M, N) \mid (M, N) \in \mathcal{TT}[\![P_2]\!]\}$$

A key feature of the definition of transition traces is that they use the reflexive-transitive closure of the one-step evaluation relation, $\to^{*}$. An important 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. Let α and β range over elements of $(\mathbf{M} \times \mathbf{M})^{*}$.

Definition **4.3** *A set* $T \subseteq \wp((\mathbf{M} \times \mathbf{M})^{+})$ *is closed under left-stuttering and absorption if it satisfies the following two conditions*

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

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

Proposition **4.4** *For all programs* P, $\mathcal{TT}[\![P]\!] = \ddagger\mathcal{TT}[\![P]\!]$.

"Stuttering" represents the fact that we can have an arbitrary interference by the context without any visible steps performed by the program, The concept and terminology are well-known from Lamport's work on temporal logics for concurrent systems [Lam91]. 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 programs termination. The "absorption" property is important because prevents "idle" computation steps from becoming semantically significant, which is a problem with the resumption approach.

4.2 A Compositional Definition of Transition Traces

We now show that the transition traces of a command can be given a denotational definition, which shows that transition trace semantics can be used to prove operational equivalence.

For the basic reaction-action pairs $(A \Leftarrow R)$, we build transition traces by simply considering all sequences of mediating transitions, followed by a terminal transition. Define the following sets[2]

$$\begin{array}{rcl} \mathsf{mediators}_{(A\Leftarrow R)} & = & \{(M,N) \mid \langle (A \Leftarrow R), M\rangle \to^* \langle (A \Leftarrow R), N\rangle\} \\ \mathsf{terminals}_{(A\Leftarrow R)} & = & \{(M,N) \mid \langle (A \Leftarrow R), M\rangle \to^* N\} \end{array}$$

Sequential composition has an easy definition. We just concatenate the transition traces from the transition traces of the components, and take their closure. Define the following sequencing operation for transition trace sets:

$$T_1 \mathbin{;} T_2 = \{\alpha\beta \mid \alpha \in T_1, \beta \in T_2\}$$

Not surprisingly parallel composition is built with the use of an interleaving combinator. For α and β in $(\mathbf{M},\mathbf{M})^*$ let $\alpha \sharp \beta$ be the set of all their interleavings, given inductively by

$$\begin{array}{rcl} \epsilon \sharp \beta & = & \beta \sharp \epsilon \quad = \quad \{\beta\} \\ (M,M')\alpha \sharp (N,N')\beta & = & \{(M,M')\gamma \mid \gamma \in \alpha \sharp (N,N')\beta\} \\ & & \cup \{(N,N')\gamma \mid \gamma \in (M,M')\alpha \sharp \beta\}\,. \end{array}$$

Now to define the transition traces of $P_1 + P_2$ we must ensure that the traces of P_1 and P_2 are interleaved, but not arbitrarily. The termination step of a parallel composition requires an agreement at the point of their termination. For this purpose, we define the following interleaving operation on transition traces:

$$T_1 \oplus T_2 = \{\alpha(M,M) \mid \alpha_1(M,M) \in T_1, \alpha_2(M,M) \in T_2, \alpha \in \alpha_1 \sharp \alpha_2\}$$

Definition 4.5 *The transition trace mapping* $\mathsf{T}[\![\_]\!] : \mathbf{P} \to \wp((\mathbf{M}\times\mathbf{M})^+)$ *is given by induction on the syntax as:*

$$\begin{array}{rcl} \mathsf{T}[\![(A \Leftarrow R)]\!] & = & (\mathsf{mediators}_{(A\Leftarrow R)})^* \mathbin{;} \mathsf{terminals}_{(A\Leftarrow R)} \\[1em] \mathsf{T}[\![P_1 \circ P_2]\!] & = & \ddagger(\mathsf{T}[\![P_2]\!] \mathbin{;} \mathsf{T}[\![P_1]\!]) \\[1em] \mathsf{T}[\![P_1 + P_2]\!] & = & \ddagger(\mathsf{T}[\![P_1]\!] \oplus \mathsf{T}[\![P_2]\!]) \end{array}$$

In Appendix A we sketch the correctness proof of the compositional definition leading to the following result:

Theorem 4.6 *For all programs P,* $\mathsf{T}[\![P]\!] = \mathcal{TT}[\![P]\!]$

[2]Note that the definitions extend to all **simple** programs, that is, programs not containing sequential composition

Define the transition trace ordering on programs $\sqsubseteq_t$ as

$$P_1 \sqsubseteq_t P_2 \iff \mathcal{TT}[\![P]\!] \subseteq \mathcal{TT}[\![P_2]\!].$$

The operations used to build the compositional definition are all monotone with respect to subset inclusion, and so a simple induction on contexts is sufficient to give

$$P_1 \sqsubseteq_t P_2 \Rightarrow \forall \mathbf{C}.\, \mathbf{C}[P_1] \sqsubseteq_t \mathbf{C}[P_2]$$

and since $P \sqsubseteq_t Q$ implies $P \leq Q$, we have the desired characterisation of operational approximation:

$$P_1 \sqsubseteq_t P_2 \Rightarrow P_1 \sqsubseteq_o P_2$$

However, as we shall show in the next section, we cannot reverse the implication—the transition trace semantics is not (inequationally) *fully abstract.*

5 Laws and Full Abstraction

Many laws of operational approximation and equivalence can now be proved using the transition trace model. In this section we state that (almost all) the partial correctness laws presented in section 2 can be shown to hold for operational approximation and equivalence, using the transition traces. However, in spite of the pedigree of Brookes' transition trace method, we show that not all laws can be verified, ie. that the semantics is not fully abstract.

5.1 Laws

We can now state a number of laws for transition trace approximation and its associated equivalence, $\equiv_t$.

PROPOSITION **5.1**

1. $P \circ (Q \circ R) \equiv_t (P \circ Q) \circ R$
2. $P + (Q + R) \equiv_t (P + Q) + R$
3. $P + Q \equiv_t Q + P$
4. $(A \Leftarrow False) + P \equiv_t P + (A \Leftarrow False) \equiv_t P$
5. $(A \Leftarrow False) \circ P \sqsubseteq_t P$
6. $P \circ (A \Leftarrow False) \equiv_t P$
7. $P \sqsubseteq_t P + P$
8. $(P_1 + P_3) \circ P_2 \sqsubseteq_t (P_1 \circ P_2) + P_3$

Note that none of the inequalities above can be strengthened to equalities. In particular, consider law 5. The transition traces of $(A \Leftarrow False)$ is the set $\ddagger\{(M, M) \mid M \in \mathbf{M}\}$, and so for all multisets N there exists a trace $\alpha(N, N) \in \mathcal{TT}[\![(A \Leftarrow False) \circ P]\!]$, which is clearly not true in general for $\mathcal{TT}[\![P]\!]$ (for a concrete example, take the one from section 3).

Many of these properties follow directly from properties of the transition trace operations. The other are proved from either the operational or compositional definitions of transition traces. A more thorough consideration of the laws is presented in [San93]

5.2 On Full Abstraction and the Big Bang

The transition trace semantics is fully abstract if transition trace approximation coincides with operational approximation. Unfortunately this does not hold, but the counterexample is interesting, and suggests modifications to either the language or the semantics for which the full abstraction question is still open.

In the construction of the transition traces of a program, when we have adjacent pairs $\cdots(M, M')(N, N')\cdots$, the change from M' to N represents changes made by the programs environment. If these changes are not realisable by an actual program then we are in danger of distinguishing between programs because of unfeasible context behaviour. First we show that when we consider the context's termination behaviour there are impossible interruptions in the transition traces. Then we give an example which shows that this leads to distinctions between operationally equivalent programs.

There are programs that always diverge. Consider any nullary reaction $(() \to A \Leftarrow True)$ which says: if the empty set is a subset of the multiset, replace that subset by A. Clearly such a reaction is always applicable, and as a consequence for all contexts $\mathbf{C}$ and programs P,

$$(() \to A \Leftarrow True) \equiv_t \mathbf{C}[(() \to A \Leftarrow True)] \sqsubseteq_t P.$$

We use this property to show that there is no "Big Bang" program which can add elements to the empty multiset and still terminate:

Proposition **5.2** $\langle P, \emptyset\rangle \to^* M \Rightarrow M = \emptyset$

PROOF Suppose that $\langle P, \emptyset\rangle \to^* M \neq \emptyset$ then there is some P' and nonempty N such that $\langle P, \emptyset\rangle \to^* \langle P', \emptyset\rangle \to \langle P', N\rangle \to^* M$. From the operational semantics we must have $P' = \mathbf{A}[(A \Leftarrow R)]$ for some active context $\mathbf{A}$ and reaction $(A \Leftarrow R)$, such that $\langle (A \Leftarrow R), \emptyset\rangle \to \langle (A \Leftarrow R), N\rangle$. Therefore the reaction must be nullary, with $R() = True$ and $A() = N$, which contradicts the assumption that the program terminates. □

Now we can see that the interruptions modeled by the transition traces of a program are potentially too liberal, since they allow sequences of the form $\cdots(M_1, \emptyset)(M_2, N_2)\cdots$ for nonempty M_2, which could never happen in an execution of the program in any context for which the *combined* system terminates. We use this fact to build a counterexample to full abstraction.

Consider the unary reaction $Await_\emptyset : (x \to x \Leftarrow True)$. This program terminates only when the multiset is empty. Now consider composition with some arbitrary program P:

Proposition **5.3** $P \circ Await_{\emptyset} \sqsubseteq_o Await_{\emptyset}$

The idea is that in any context, if either of these programs terminates it must be the case that at some intermediate point $Await_{\emptyset}$ terminates. Since this can only happen when the multiset becomes empty, it follows from proposition 5.2 that any computation following this point cannot change the multiset and still terminate. Therefore, the additional composition with P can never alter the result of a terminating computation, which must always be the empty multiset, but *can* make the program terminate less often.

Proposition **5.4** $\mathcal{TT}\llbracket\_\rrbracket$ *is not fully abstract.*

proof Take $P = (A \Leftarrow False)$. Then

$$\mathcal{TT}\llbracket(A \Leftarrow False) \circ Await_{\emptyset}\rrbracket = \ddagger\,\{(\emptyset,\emptyset)(M,M) \mid M \in \mathbf{M}\}$$

but this is not a subset of $\mathcal{TT}\llbracket Await_{\emptyset}\rrbracket = \ddagger\,\{(\emptyset,\emptyset)\}$. □

We postpone further discussion of the full abstraction problem until the concluding section.

6 New Combining Forms

In this section we consider the addition of other combining forms for programs that provide a conservative extension of transition trace equivalence, and relationships between them.

Vanilla Parallel Composition The more usual form of parallel program composition does not require that the two programs terminate synchronously. Extend the syntax of the language with $P ::= P_1 \| P_2$, and the active contexts with $\mathbf{A} ::= \mathbf{A}\|P \mid P\|\mathbf{A}$. Now we can give the rules for one step evaluation:

$$\frac{\langle Q, M\rangle \to M}{\langle P\|Q, M\rangle \to \langle P, M\rangle} \qquad \frac{\langle P, M\rangle \to M}{\langle P\|Q, M\rangle \to \langle Q, M\rangle}$$

The transition traces for $P\|Q$ can be given compositionally just by (taking the closure of) interleaving those of P with those of Q, and the proof extends easily. The expected associativity and commutativity properties also hold for $\|$. Some relationships with $\circ$ and $+$ can be summarised in the following diagram where $\mathsf{FA} = (A \Leftarrow False)$ and the arrows ($\to$) depict the ordering $\sqsubseteq_t$.

$$\begin{tikzcd}
 & & (\mathsf{FA} \circ P) + (\mathsf{FA} \circ Q) & & \\
 & & P\|Q \arrow[u] & & \\
 & (\mathsf{FA} \circ P) + Q \arrow[ur] & & P + (\mathsf{FA} \circ Q) \arrow[ul] & \\
Q \circ P \arrow[ur] & & P + Q \arrow[ul] \arrow[ur] & & P \circ Q \arrow[ul]
\end{tikzcd}$$

Nondeterministic Choice A nondeterministic choice operator is also easily added, just as in [Bro93]. Extend $\mathbf{P} ::= P_1 \vee P_2$. Evaluation is specified by insisting that the choice is committed before any evaluation can occur. Therefore we do not extend the active contexts, but have the two rules:

$$\langle P \vee Q, M\rangle \to \langle P, M\rangle \qquad \langle P \vee Q, M\rangle \to \langle Q, M\rangle$$

Again the extension of the semantics is straightforward. The transition traces are modeled compositionally using set union, and $+$ and $\circ$ distribute over $\vee$. Perhaps more interestingly we get an exact characterisation of the above form of parallel composition:

$$P \| Q \equiv_t ((\mathsf{FA} \circ P) + Q) \vee (P + (\mathsf{FA} \circ Q))$$

Critical Region There are problems regarding language extensions when we attempt to add operators which allow the atomic steps to have a larger granularity. The problem is that the definition of $+$ requires, for termination, that both operands must agree. This demands that terminal transitions must be of the form $\langle P, M\rangle \to M$, otherwise implementation of parallel composition would require multiset duplication and possible backtracking of multiset computations.

However, some control over atomicity seems desirable to guarantee interference freedom. With the above remarks in mind we give the definition of a form of guarded *critical region* $P \lhd Q$, which allows for the uninterrupted execution of sub-program P as soon as Q terminates. To do this we extend active contexts to also include contexts of the form $P \lhd \mathbf{A}$, and define the following intermediate transition as follows:

$$\frac{\langle Q, M\rangle \to M \quad \langle P, M\rangle \to^* N}{\langle P \lhd Q, M\rangle \to \langle \mathsf{FA}, N\rangle}$$

The use of the "dummy" program FA in the right hand side of this transition means that a critical region construct fits with parallel composition in a computationally reasonable way. The price that is paid for this is that $\lhd$ is not associative (the compositional definition of its transition traces is left as an exercise).

7 Conclusions

We have presented a compositional semantics for a number of combining forms for the Gamma model of multiset programming, beginning with those defined in [HMS92]. The semantics is useful for verifying substitutional refinement laws, and thus extend the applicability of some of the transformation techniques discussed in previous work. It is also a useful tool for studying the impact of language extensions. In the remainder of the paper we consider areas of future work.

In section 5, the fact that a program cannot add elements to an empty multiset and still terminate was used to show that the semantics is not fully abstract. This suggests a modification to the semantics to only allow feasible transition traces for which adjacent pairs $(M_i, N_i)(M_{i+1}, N_{i+1})$ satisfy the property $N_i = \emptyset \Rightarrow M_{i+1} = \emptyset$. It is an open question whether this modification leads to a fully abstract model.

Alternatively, we might view the above property as an indication of a lack of expressive power (since, for example, it implies that the constant programs $K_M : \forall N. \langle K_M, N\rangle \rightarrow^* M$ are not definable) and look for suitable extensions. One such extension involves the addition of reactions which are parameterised by a *critical mass* limiting the global size of the multisets in which they are applicable[3]. The critical mass reactions are sufficient to define the family of programs $\{Await_M\}$ which terminate if and only if the multiset becomes equal to M (cf. Hennessy and Plotkin's *await* commands). If we also include the guarded critical region construct suggested in the previous section, they can be used to give a full abstraction proof, directly adapting the method used by Brookes.

In spite of the absence of full abstraction, the transition traces are sufficiently abstract to prove the expected laws. A more detailed study of the laws of the parallel and sequential composition is given in [San93], in which we verify or refute all of the laws for the simple relational operational ordering given in [HMS92], including the "residual program" laws. In addition it is shown that every program is refined by a product ($\circ$) of sums ($+$) of basic reactions.

Acknowledgements Thanks to Chris Hankin and Daniel Le Métayer for numerous discussions and ideas. Thanks to Steve Brookes, Samson Abramsky and Geoff Burn for helpful discussions and advice.

A Proof of the compositional definition of the transition traces

In this appendix the proof of correctness for the compositional definition of transition traces is sketched.

LEMMA **A.1** *$\mathsf{T}[\![P]\!]$ is closed.*

LEMMA **A.2**

1. $\langle P, M\rangle \rightarrow^* N \Rightarrow (M, N) \in \mathsf{T}[\![P]\!]$
2. $\langle P, M\rangle \rightarrow^* \langle P', M'\rangle \;\&\; \alpha \in \mathsf{T}[\![P']\!] \Longrightarrow (M, N)\alpha \in \mathsf{T}[\![P]\!]$

PROOF Prove that the lemma holds for one-step evaluation (inductions in the structure of P), and extend to $\rightarrow^*$ by induction on the length of the reflexive-transitive closure of $\rightarrow$ using lemma A.1. □

[3]Although this would violate the "locality principal" [BM93] of the Gamma model.

Lemma **A.3**

1. $\beta_1 \in \mathcal{TT}[\![P_1]\!]$ & $\beta_2 \in \mathcal{TT}[\![P_2]\!] \Longrightarrow \beta_2\beta_1 \in \mathcal{TT}[\![P_1 \circ P_2]\!]$
2. $\alpha(M,M) \in \mathcal{TT}[\![P_1]\!]$ & $\beta(M,M) \in \mathcal{TT}[\![P_2]\!]$ & $\gamma \in \alpha \sharp \beta \Longrightarrow \gamma(M,M) \in \mathcal{TT}[\![P_1 + P_2]\!]$

PROOF Inductions on the lengths of β_2 and γ respectively. □

Theorem **A.4** *For all programs P, $\mathcal{TT}[\![P]\!] = \mathsf{T}[\![P]\!]$.*

PROOF We prove it in two halves:

- $\mathcal{TT}[\![P]\!] \subseteq \mathsf{T}[\![P]\!]$: The (implicitly inductive) definition of $\mathcal{TT}[\![\_]\!]$ can be given as the least fixed point of the monotonic functional $\mathrm{F} : (\mathbf{P} \to \wp((\mathbf{M} \times \mathbf{M})^*)) \to \mathbf{P} \to \wp((\mathbf{M} \times \mathbf{M})^*)$
$$\begin{aligned}\mathrm{F} = \lambda R.\lambda P.\ &\{(M,N) \mid \langle P, M\rangle \to^* N\} \\ &\cup \{(M,N)\alpha \mid \langle P,M\rangle \to^* \langle P', N\rangle, \alpha \in R(P')\}\end{aligned}$$
From lemma A.2 we immediately have that $\mathsf{T}[\![\_]\!]$ is a post fixed point of F and so by the Knaster-Tarski fixed point theorem contains the least fixed point, $\mathcal{TT}[\![\_]\!]$.

- $\mathsf{T}[\![P]\!] \subseteq \mathcal{TT}[\![P]\!]$: By induction on the structure of P we show that $\delta \in \mathsf{T}[\![P]\!] \Rightarrow \delta \in \mathcal{TT}[\![P]\!]$. The base case ($P = (A \Leftarrow R)$) is obvious. In both the inductive cases we assume without loss of generality that δ comes from
$$\{\alpha_2\alpha_1 \mid \alpha_1 \in \mathsf{T}[\![P_1]\!], \alpha_2 \in \mathsf{T}[\![P_2]\!]\}\ ,\ \text{and}$$
$$\{\alpha(M,M) \mid \alpha_1(M,M) \in \mathsf{T}[\![P_1]\!], \alpha_2(M,M) \in \mathsf{T}[\![P_2]\!], \alpha \in \alpha_1 \sharp \alpha_2\}$$
respectively (ie. it is sufficient to ignore the closure of these sets in the compositional definitions).

  Case ($P = P_1 \circ P_2$): $\delta = \delta_2\delta_1$ for some $\delta_1 \in \mathsf{T}[\![P_1]\!], \delta_2 \in \mathsf{T}[\![P_2]\!]$. The induction hypothesis gives that $\delta_1 \in \mathcal{TT}[\![P_1]\!], \delta_2 \in \mathcal{TT}[\![P_2]\!]$, and the result follows from lemma A.3.

  Case ($P = P_1 + P_2$): $\delta = \gamma(N,N)$ for some $\gamma \in \alpha \sharp \beta$, $\alpha(N,N) \in \mathsf{T}[\![P_1]\!]$ and $\beta(N,N) \in \mathsf{T}[\![P_2]\!]$. From the induction hypothesis we know that $\alpha(N,N) \in \mathcal{TT}[\![P_1]\!]$, $\beta(N,N) \in \mathcal{TT}[\![P_2]\!]$ and the result follows from lemma A.3.

□

References

[Abr79] K. Abrahamson. Modal logic of concurrent nondeterministic programs. In *Proceedings of the International Symposium on Semantics of Concurrent Computation*, volume 70, pages 21–33. Springer-Verlag, 1979.

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

[BM93] J.-P. Banâtre and D. Le Métayer. Programming by multiset transformation. *CACM*, January 1993. (INRIA research report 1205, April 1990).

[Bro93] S. Brookes. Full abstraction for a shared variable parallel language. In *Logic In Computer Science*, 1993. (to appear).

[HMS92] C. Hankin, D. Le Métayer, and D. Sands. A calculus of Gamma programs. Research Report DOC 92/22 (28 pages), Department of Computing, Imperial College, 1992. (short version to appear in the Proceedings of the Fifth Annual Workshop on Languages and Compilers for Parallelism, Aug 1992, Springer-Verlag).

[HP79] M. Hennessy and G. D. Plotkin. Full abstraction for a simple parallel programming lanuage. In *Mathematical Foundations of Computer Science*, volume 74 of *LNCS*, pages 108–120. Springer-Verlag, 1979.

[Lam91] L. Lamport. The Temporal Logic of Actions. Technical Report 79, DEC Systems Research Center, Palo Alto, CA, 1991.

[San93] D. Sands. Laws of parallel synchronised termination. In *Theory and Formal Methods 1993: Proceedings of the First Imperial College, Department of Computing, Workshop on Theory and Formal Methods*, Isle of Thorns, UK, 1993. Springer-Verlag Workshops in Computer Science.