Composed Reduction Systems

David Sands*
DIKU, University of Copenhagen†

Abstract

This paper studies *composed reduction systems*: a system of programs built up from the reduction relations of some reduction system, by means of parallel and sequential composition operators. The trace-based compositional semantics of composed reduction systems is considered, and a new graph-representation is introduced as an alternative basis for the study of compositional semantics, refinement, true concurrency (in the case of composed rewriting systems) and program logics.

1 Introduction

Reduction systems are simply sets equipped with some collection of binary "rewrite" relations. A reduction systems can be thought of as an abstract view of computation, embodying the fundamental computational concepts of *iteration*, *termination*, and *nontermination*. Computation is the process of repeatedly rewriting, beginning with some object of the set, and termination corresponds to obtaining an object which cannot be rewritten further; nontermination is the ability to rewrite indefinitely.

Since reduction systems have little structure, there are relatively few properties one can state about theses systems, although unique-termination ("Church-Rosser") properties of reduction systems have been studied by eg. Rosen [Ros73], Hindley [Hin69] Staples [Sta74].

In this paper we consider systems ("programs") whose basic components are the reduction relations of some reduction system. These systems, which we call *composed reduction systems*, are built by composing reduction relations with two natural composition operators: *parallel* and *sequential* composition. Composed reduction systems are not necessarily reduction systems, but they possess a notion of a "reduction step", and a corresponding notion of termination.

- Parallel composition allows arbitrary interleaving of reduction steps. In the simplest case, the parallel composition of two reduction relations corresponds to the union of these relations. Parallel composition terminates when, simultaneously, both sub-systems have terminated.

*This work was partially funded by ESPRIT BRA 9102, "Coordination"
†Universitetsparken 1, 2100 København Ø, DENMARK. e-mail: dave@diku.dk

- Sequential composition, on the other hand, takes us outside the realm of reduction systems (over the given set). The sequential composition of two reduction relations is the system which behaves like the first reduction relation, until termination of the first system, after which it behaves like the second system. The sequentially composed system is said to terminate when the second sub-system has terminated.

Note, then, that the sequential composition of two reduction relations (the simplest case) is not the relational composition of these relations. Composed reduction systems over a given reduction system are built from arbitrary sequential and parallel compositions of reduction relations.

In this paper we study the semantics of composed reduction systems, expressed in terms of its constituent reduction relations. We focus on a comparison relation for programs which partially orders programs on the basis of their "input-output" behaviours, and is also a precongruence with respect to program construction.

In the first part of the paper (Section 2) we consider a standard compositional semantics based on "reactive traces" (sequences of object-pairs) derived from the SOS-like rules which give the operational semantics for composed reduction systems. We outline some of the program laws that can be obtained, and consider the relationship to an alternative form of parallel composition.

In the second part of the paper (Section 3), we define a static graph representation for programs, and argue that it forms a better basis (than the transition traces) for the study of:

- compositional semantics, since it is higher-level than the transition traces;
- refinement laws, since the graphs can also be defined compositionally;
- concurrency, since "concurrently active" reduction relations are explicit in the representation, and
- program logics, since logics for the underlying reduction systems can be used to reasoning about paths through the graph.

Related Work and Applications This work grew out of the study of composition of specific kind of reduction system, namely programs in the Gamma model [BM93], which can be thought of as conditional associative-commutative string rewriting. The composition operators for Gamma were introduced in [HMS92], and the compositional semantics and laws were studied in [San93a][San93b]. The development of section 2 is a direct (and straightforward) adaptation of [San93a][San93b] to this more general setting. The graph representation in section 3 is new, and is particularly relevant from the point of view of composed Gamma programs. [1]

The techniques given here may also be interesting when applied to other concrete reduction systems. In particular we have in mind *rewriting systems*

[1] For example, through this representation have discovered additional laws for Gamma programs.

in which the objects rewritten have some structure (eg. trees, graphs, strings), and the reduction relation is specified by rules for rewriting a substructure, in terms of purely *local* conditions. For such systems (eg. the usual notion of term rewriting [Klo92] [DJ89]) there is a natural (implicit) notion of concurrency, viz. disjoint parts of a substructure can be rewritten asynchronously, and hence concurrently. This view of rewriting as a natural vehicle for concurrency and parallel programming is central to Meseguer's approach [Mes92][MW91]; the composition operators studied here also make sense in that setting.

Another form of reduction system, where one could reasonably employ the composition operators studied here, is the guarded iteration statement from [Dij76], also known as *action systems* [Bac89a][Bac89b]. Action systems are nondeterministic **do-od** programs consisting of a collection of guarded atomic actions, which are executed nondeterministicly so long as some guard remains true. In their uninitialised form, guarded iteration statements can be thought of as reduction systems over program states. The method of parallel execution is to allow actions involving disjoint program variables to be executed in parallel, which is consistent with the rewriting viewpoint above. In [Bac89b] Back studies compositional notions of refinement for action systems with respect to a meta-linguistic parallel composition operator. [2] The parallel composition studied here is strictly more general since it permits parallel composition of sequentially composed systems.

2 Operational and Compositional Semantics

In this section we give the operational semantics of composed reduction systems built from basic reduction relations, parallel and sequential composition.

In what follows, we assume some reduction system $\langle \mathbf{U}, \{\rightarrow_{\mathbf{r}}\}_{\mathbf{r}\in\mathcal{R}}\rangle$, where $\mathbf{U}$ is a set, with typical elements $M, N, M_1 \ldots$. We will sometimes refer to the elements of $\mathbf{U}$ as *states*. The reduction relations, $\{\rightarrow_{\mathbf{r}}\}_{\mathbf{r}\in\mathcal{R}}$ are just binary relations on states.We will think of the elements of the indexing set $\mathcal{R}$, ranged over by $\mathbf{r}, \mathbf{r}_2 \ldots$, as the basic units of our composed reduction systems. Somewhat improperly, for more concrete examples we will think of $\mathcal{R}$ as the set of *representations* of the corresponding reduction relation.

With respect to some $\mathbf{r}$, we say that

- M reduces to N if $M \rightarrow_{\mathbf{r}} N$ (ie. $(M, N) \in \rightarrow_{\mathbf{r}}$);
- M converges immediately, written $M{\downarrow}^{\mathbf{r}}$, if $\neg\exists N.\, M \rightarrow_{\mathbf{r}} N$.

For the moment we consider composed reduction systems, ranged over by P, Q, P_1, Q_1 etc, given by the following grammar:

$$P ::= \mathbf{r} \mid P \,;\, Q \mid P \mathbin{+\!\!\!\!\!\!\;\,+} Q$$

Henceforth we will use the terms "composed reduction system" and "program" synonymously.

[2] UNITY [CM88] has a similar composition operator, called *union*, but UNITY is not a reduction system in the same sense because the notion of termination for UNITY is that of *stability*—reaching a fixed-point—rather than inactivity.

2.1 SOS semantics

Because of the presence of sequential composition, programs cannot be viewed as reduction systems over $\mathbf{U}$, since the program is not a static entity. To define the semantics for these programs we define a single step transition relation between *configurations*. The configurations are program-state pairs, written $\langle P, M\rangle$. The final result of a computation is given by an immediate-convergence predicate, $\downarrow$, on configurations. Single step reduction and immediate-convergence is given by SOS-style rules in figure 1. It is easily verified that immediate con-

$$\frac{M \to_{\mathbf{r}} N}{\langle \mathbf{r}, M\rangle \to \langle \mathbf{r}, N\rangle} \qquad \frac{M{\downarrow}^{\mathbf{r}}}{\langle \mathbf{r}, M\rangle{\downarrow}}$$

$$\frac{\langle P, M\rangle \to \langle P', M'\rangle}{\langle P \,;\, Q, M\rangle \to \langle P \,;\, Q, M'\rangle} \qquad \frac{\langle P, M\rangle{\downarrow}}{\langle P \,;\, Q, M\rangle \to \langle Q, M\rangle}$$

$$\frac{\langle P, M\rangle \to \langle P', M'\rangle}{\langle P \mathbin{\#} Q, M\rangle \to \langle P' \mathbin{\#} Q, M'\rangle} \quad \frac{\langle Q, M\rangle \to \langle Q', M'\rangle}{\langle P \mathbin{\#} Q, M\rangle \to \langle P \mathbin{\#} Q', M'\rangle} \quad \frac{\langle P, M\rangle{\downarrow} \quad \langle Q, M\rangle{\downarrow}}{\langle P \mathbin{\#} Q, M\rangle{\downarrow}}$$

Figure 1: Structural Operational Semantics of composed reduction systems

vergence of a configuration corresponds to the absence of any transitions for that configuration. In other words, $\langle P, M\rangle \to \langle Q, N\rangle$ for some $\langle Q, N\rangle$ if and only if $\neg(\langle P, M\rangle{\downarrow})$.

Let $\to^*$ denote the transitive, reflexive closure of $\to$. By a small abuse of the notation, we will write $\langle P, M\rangle \to^* N$ to mean that there exists some $\langle Q, N\rangle$ such that $\langle P, M\rangle \to^* \langle Q, N\rangle$ and $\langle Q, N\rangle{\downarrow}$.

2.2 Behavioural orderings

In this paper we will focus on the relational (input-output) behaviours of a program. 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. Non-termination, 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 $\langle P_i, M_i\rangle \to \langle P_{i+1}, M_{i+1}\rangle$.*

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 nontermination, which we represent as a possible "output" using symbol '$\perp$' ($\notin \mathbf{U}$):

Definition 2 *The behaviours of a program P are given by*

$$\begin{array}{rcl}\mathcal{B}(P) & = & \{(M, N) \mid \langle P, M\rangle \to^* N\} \\ & \cup & \{(M, \perp) \mid \langle P, M\rangle{\uparrow}\}\end{array}$$

Note that for every P, M, either $(M, N) \in \mathcal{B}(P)$ for some N, or $(M, \bot) \in \mathcal{B}(P)$ (or both). There are a variety of orderings on programs obtained by comparing their behaviours: the *partial correctness* ordering ignores divergent behaviours; the *lower* and *upper* orderings are formed by considering the associated discrete power-domain orderings on $\mathbf{U}_\bot$. In this study we only consider the *strong correctness* ordering, which attaches the same significance to nonterminating computations as to the terminating ones. The strong correctness ordering is defined to be the largest (pre)congruence which satisfies

$$P \sqsubseteq_o Q \Rightarrow \mathcal{B}(P) \subseteq \mathcal{B}(Q)$$

This is given directly by the following:

Definition 3 *Let* $\mathbf{C}$ *range over program contexts. We define* strong precongruence *(*$\sqsubseteq_o$*) and* strong congruence*(*$\equiv_o$*) respectively by:*

$$\begin{array}{lcl} P \sqsubseteq_o Q & \iff & \forall \mathbf{C}. \mathcal{B}(\mathbf{C}[P]) \subseteq \mathcal{B}(\mathbf{C}[Q]) \\ P \equiv_o Q & \iff & P \sqsubseteq_o Q \;\&\; Q \sqsubseteq_o P \end{array}$$

2.3 Laws

In this section we present a number of the basic laws of strong precongruence, and show the relationship to an alternative definition of parallel composition. Let Δ denote the empty reduction relation, satisfying $\forall M. M{\downarrow}^{\Delta}$. For example, in conditional rewriting systems, this could be represented by a reduction rule with the condition *false*.

Proposition 4

1. $P \,;\, (Q \,;\, R) \equiv_o (P \,;\, Q) \,;\, R$	5. $\Delta \mathbin{\#} P \equiv_o P \mathbin{\#} \Delta \equiv_o P$
2. $P \mathbin{\#} (Q \mathbin{\#} R) \equiv_o (P \mathbin{\#} Q) \mathbin{\#} R$	6. $P \sqsubseteq_o P \,;\, \Delta$
3. $P \mathbin{\#} Q \equiv_o Q \mathbin{\#} P$	7. $P \equiv_o \Delta \,;\, P$
4. $Q \,;\, (P_1 \mathbin{\#} P_2) \sqsubseteq_o (Q \,;\, P_1) \mathbin{\#} P_2$	8. $P \sqsubseteq_o P \mathbin{\#} P$

These are just a few of the laws of strong precongruence. In fact, almost all of the partial correctness laws of composed Gamma programs [San93b] hold for these more general composed reduction systems. Note in particular that law 6 cannot be strengthened to an equality, ie.

$$P \,;\, \Delta \not\equiv_o P$$

The intuition for this is that Δ acts as a de-synchroniser for parallel composition: P must synchronise with its context in order to terminate, but with $P \,;\, \Delta$, P is allowed to terminate autonomously, leaving Δ to synchronise with its context—which it is trivially always able to do. As an example consider the following two term rewrite rules, where a and b are constants: $a \to b$ and $b \to a$. It is easily seen that $\langle (a \to b) \mathbin{\#} (b \to a), a \rangle$ can never terminate, but

$$\langle (P_1 \,;\, \Delta) \mathbin{\#} P_2, a \rangle \to^* a$$

and so $(P_1 \,;\, \Delta) \mathbin{\#} P_2 \not\sqsubseteq_o P_1 \mathbin{\#} P_2$.

An Alternative Parallel Composition

There is a natural alternative form of parallel program composition, which does not require that the two programs terminate synchronously. Extend the syntax of the language with $P ::= P_1 \,\|\, P_2$, and add the operational rules:

$$\frac{\langle Q, M\rangle\!\downarrow}{\langle P \,\|\, Q, M\rangle \rightarrow \langle P, M\rangle} \qquad \frac{\langle P, M\rangle\!\downarrow}{\langle P \,\|\, Q, M\rangle \rightarrow \langle Q, M\rangle}$$

The expected associativity and commutativity properties also hold for $\|$. Some relationships with ; and $\#$ can be summarised in the diagram in figure 2, where the arrows ($\rightarrow$) depict the ordering $\sqsubseteq_o$. To give some intuition to the fact in

$$\begin{array}{ccccc}
 & & (P \,;\, \Delta) \mathbin{\#} (Q \,;\, \Delta) \equiv_o (P \,;\, \Delta) \,\|\, (Q \,;\, \Delta) & & \\
 & & \uparrow & & \\
 & & P \,\|\, Q & & \\
 & \nearrow & & \nwarrow & \\
 & (P \,;\, \Delta) \mathbin{\#} Q & & P \mathbin{\#} (Q \,;\, \Delta) & \\
 \nearrow & & \nwarrow \quad \nearrow & & \nwarrow \\
 P \,;\, Q & & P \mathbin{\#} Q & & Q \,;\, P
\end{array}$$

Figure 2: Relationships between Compositions

this diagram, consider the increasing chain of "behaviours":

$$P \,;\, Q \sqsubseteq_o (P \,;\, \Delta) \mathbin{\#} Q \sqsubseteq_o P \,\|\, Q \sqsubseteq_o (P \,;\, \Delta) \mathbin{\#} (Q \,;\, \Delta)$$

For $P \,\|\, Q$ to terminate, either P (or some derivative thereof) terminates followed by Q, or vice-versa, and either P or Q is left to synchronise its termination with the context. The system $(P \,;\, \Delta) \mathbin{\#} Q$ has fewer behaviours because although reductions from P are potentially concurrent with reductions from Q, P must always terminate first. $(P \,;\, \Delta) \mathbin{\#} (Q \,;\, \Delta)$, on the other hand, exhibits more behaviours since P and Q are "concurrent", can terminate autonomously, and neither of them is required to synchronise its termination with the context.

2.4 Trace Semantics

We can characterise $\sqsubseteq_o$ (in order to *prove* the laws of the previous section) by finding a compositional semantics which is consistent (ie. sound) with respect to the behaviours. Clearly the *behaviours* of a program will not suffice as its denotation. As is well-known from the study state-based concurrency, it insufficient to use sequences of states as a means of distinguishing programs. The solution we adopt follows a simple approach to modelling shared-state (interleaving) concurrency via sequences of state-pairs[3] (eg. sequences of "moves"[Abr79];

[3]The set of all such sequences for a program can be thought of as an "unraveling" of the program's *resumption semantics* [Plo76][HP79]. This "unraveling" leads to a mathematically simpler domain (no powerdomains) which is more amenable to further refinements than resumptions.

"abstract paths" of [Par79]). Following the terminology of [Bro93], we will use the term *transition traces*, or simply *traces* to refer to this kind of sequence. In these models, a pair of states in the trace of a program represents an atomic computation step of the program; adjacent pairs in any given sequence model a possible interference by some other process executing in parallel with the program.

The transition traces have a straightforward operational specification:

Definition 5 *The transition traces of a program,* $\mathsf{T}[\![P]\!]$, *are the finite and infinite sequences of state-pairs, given by:*

$$\begin{array}{rcl} \mathsf{T}[\![P]\!] & = & \{(M_0, N_0)(M_1, N_1)\ldots(M_k, N_k) \mid \\ & & \langle P, M_0\rangle \rightarrow \langle P_1, N_0\rangle \;\&\; \\ & & \langle P_1, M_1\rangle \rightarrow \langle P_2, N_1\rangle \;\&\; \ldots \;\&\; \langle P_k, M_k\rangle{\downarrow}, \;\&\; M_k = N_k\} \\ & \cup & \\ & & \{(M_0, N_0)(M_1, N_1)\ldots(M_i, N_i)\ldots \mid \\ & & \langle P, M_0\rangle \rightarrow \langle P_1, N_0\rangle \;\&\; \langle P_i, M_i\rangle \rightarrow \langle P_{i+1}, N_i\rangle, i \geq 1\} \end{array}$$

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

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

represents computation steps of program P in some context; starting with state M_0, each of the pairs (M_i, N_i) represents computation steps performed by (derivatives of) P and the adjacent states N_{i-1}, M_i represent possible interfering computation steps performed by the "context". If the trace is finite then the last step corresponds to the termination "step" for a derivative of P.

Stuttering

Clearly the behaviours of a program are obtainable from its transition traces, by considering the "chained" traces of the form: $(M_0, M_1)(M_1, M_2)\ldots$. Transition traces are adequate for giving a compositional semantics to composed reduction systems, by interpreting sequential composition as (set-wise) trace concatenation, and parallel composition as interleaving (with the proviso that interleaved finite traces must agree on their last elements).

However, the transition traces distinguish between programs which compute at different "speeds". For example, considering the empty action system $DO \;\; OD$, then the transition traces of $DO \;\; OD$ are different from those of $(DO \;\; OD)$; $(DO \;\; OD)$ [4] The key to obtaining a better level of abstraction is to equate processes which only vary by "uninteresting" steps. This is the "stuttering equivalence" well-known from Lamport's work on temporal logics for concurrent systems [Lam89]. Closure under stuttering equivalence has been used by de Boer et al [dBKPR91], and by Brookes [Bro93] to provide fully abstract semantics for languages with shared-state and parallel composition.

Following Brookes [Bro93] we define a closure operation for sets of transition traces:

[4] In order to obtain full abstraction for a while-language with parallel composition, Hennessy and Plotkin added a co-routine command, which is able to distinguish these programs.

Definition 6 *Let ϵ denote the empty sequence. Let α range over finite sequences of state pairs, and β range over finite or infinite sequences. A set T of finite and infinite traces is closed under left-stuttering*[5] *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.

In [dBKPR91] a slightly different closure operation is used, in which only stuttered steps can be absorbed. With respect to the above closure conditions, the difference is that in the clause for absorption we should also require that either $M = N$ or $N = M'$. This leads to a coarser abstraction for specific reduction systems; for example, the composed string-rewriting systems: $(1, 1 \rightarrow 2, 2) \mathbin{+\!\!\!+} (1 \rightarrow 2)$ and $(1 \rightarrow 2)$ have different traces under the stuttering-closure operation of [dBKPR91], but are the same under $\ddagger$.

Clearly the behaviours are also derivable from $\ddagger\mathsf{T}[\![P]\!]$, and what is more, $\ddagger\mathsf{T}[\![P]\!]$ can be specified compositionally (using monotonic operators) which gives the following:

Proposition 7 $\ddagger\mathsf{T}[\![P]\!] \subseteq \ddagger\mathsf{T}[\![Q]\!] \Longrightarrow P \sqsubseteq_o Q$

In the appendix we give the compositional definition of transition traces.

For a specific collection of reduction relations over some given universe, the transition traces may not be *fully abstract.* In other words, we cannot reverse the implication in the above proposition. For a specific example where full abstraction fails, see [San93a]. Even if we allow all reduction relations over a given universe it is unclear as to whether the transition traces are fully abstract.

3 Graph Representation

In this section we outline a static graph-representation for composed reduction systems, and argue that it forms a better basis for the study of compositional semantics, refinement, true concurrency and program logics.

The graph representation we will develop is something like a finite, acyclic control-flow graph, where each node corresponds to a simple form of loop. A node carries a set of reduction relations which are (con)currently active; an edge represents an internal termination step, where the child node may inherit some reductions from the parent but adds some new active reductions. From the viewpoint of the observational semantics, we will identify the graph with its set of complete paths.

The idea is best illustrated with an example. Consider a program consisting of four reduction relations:

$$(\mathbf{r}_1 \,;\, \mathbf{r}_2) \mathbin{+\!\!\!+} (\mathbf{r}_3 \,;\, \mathbf{r}_4)$$

[5] 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.

Initially $\mathbf{r}_1$ and $\mathbf{r}_3$ are *active* and thereby able to contribute to the reduction steps. At some time, $\mathbf{r}_1$ or $\mathbf{r}_3$ may be able to terminate. Suppose $\mathbf{r}_1$ terminates first; then $\mathbf{r}_2$ and $\mathbf{r}_3$ become active. Symmetrically, if $\mathbf{r}_3$ terminates first then $\mathbf{r}_1$ and $\mathbf{r}_4$ become active. Continuing in this way we construct the graph for this program:

$$\begin{tikzcd}
 & \{\mathbf{r}_1, \mathbf{r}_3\} \arrow[dl, "\{\mathbf{r}_3\}"'] \arrow[dr, "\{\mathbf{r}_1\}"] & \\
\{\mathbf{r}_1, \mathbf{r}_4\} \arrow[dr, "\{\mathbf{r}_1\}"'] & & \{\mathbf{r}_2, \mathbf{r}_3\} \arrow[dl, "\{\mathbf{r}_3\}"] \\
 & \{\mathbf{r}_2, \mathbf{r}_4\} &
\end{tikzcd}$$

The operational semantics of such graphs should be transparent: control begins at the root-node of the graph, and each node is labeled with a set of concurrently active reduction-relations; each arc is labeled with a set of reduction relations which must converge with respect to the current state for the control to be allowed move along that edge.

3.1 From SOS rules to Graph Representation

The graph representation will be constructed from two "abstract interpretations" of the one-step evaluation relation. Consider any possible one-step reduction on configurations:

$$\langle P, M\rangle \rightarrow \langle P', M'\rangle$$

From inspection of the rules it is clear that either:

1. $P \neq P'$ and $M = M'$, or

2. $P = P'$ and $M \rightarrow_{\mathbf{r}} M'$ for some reduction $\mathbf{r}$ in P.

In the terminology of [HMS92], we call a transition of the first kind as a *passive* step, and one of the second kind as an *active* step. The passive step corresponds to some internal termination step in which the left-operand of a sequential composition is discarded. The convergence of a configuration can similarly be considered to be a passive step. An active step corresponds to a reduction step on the state-component of a configuration.

We construct the graph representation of a given program by separately *abstracting*:

1. the passive steps, which will give us the arcs in the graph, and

2. the active steps which will tell us what reductions are contained in the nodes.

Abstract Passive Steps We abstract the passive steps performable by a program via a (labeled) transition system with judgements of the form $P \overset{R}{\rightsquigarrow} Q$, where R is a set of reductions. As an auxiliary, we define a notion of convergence for programs which is an abstraction of the convergence predicate for configurations. The abstract convergence predicate is trivial: a program can converge only if it does not contain any sequential compositions. Let $[P]$ denote the set of reduction relations that comprise the program P. Figure 3 defines the rules, closely following the form of the rules of figure 1.

$$\frac{}{\mathbf{r}\!\downarrow}$$

$$\frac{P \overset{R}{\rightsquigarrow} P'}{P\,;Q \overset{R}{\rightsquigarrow} P'\,;Q} \qquad \frac{P\!\downarrow}{P\,;Q \overset{[P]}{\rightsquigarrow} Q}$$

$$\frac{P \overset{R}{\rightsquigarrow} P'}{P \,\|\, Q \overset{R}{\rightsquigarrow} P' \,\|\, Q} \qquad \frac{Q \overset{R}{\rightsquigarrow} Q'}{P \,\|\, Q \overset{R}{\rightsquigarrow} P \,\|\, Q'} \qquad \frac{P\!\downarrow \quad Q\!\downarrow}{P \,\|\, Q\!\downarrow}$$

Figure 3: Abstract Passive Steps

Active Region We abstract the active steps of a program simply by saying which reductions in the program are immediately applicable. The immediately-applicable reductions are just those which are not "guarded" by a sequential composition on their left. The *active region* of a program P, written $\lceil P \rceil$, is defined inductively by:

$$\begin{array}{rcl} \lceil \mathbf{r} \rceil & = & \{\mathbf{r}\} \\ \lceil P\,;Q \rceil & = & \lceil P \rceil \\ \lceil P \,\|\, Q \rceil & = & \lceil P \rceil \cup \lceil Q \rceil \end{array}$$

The following proposition states the precise relationship between the above abstractions and the transition relation of the structural operational semantics:

Proposition 8 *For all composed reduction systems P over some universe $\mathbf{U}$, and for all $M, N \in \mathbf{U}$,*
$\langle P, M \rangle \to \langle Q, N \rangle$ *if and only if either*

1. *$M = N$ and $P \overset{R}{\rightsquigarrow} Q$ for some R such that for all $\mathbf{r} \in R$, $M\!\downarrow^{\mathbf{r}}$, or*

2. *$P = Q$, and there exits some $\mathbf{r} \in \lceil P \rceil$ such that $M \to_{\mathbf{r}} N$.*

The graph form will be constructed by combining the passive steps with the active regions. We note the following facts about the passive steps.

- Passive steps are normalising: ie. there are no infinite chains of the form $P \overset{R_1}{\rightsquigarrow} P_1 \overset{R_2}{\rightsquigarrow} P_2 \overset{R_3}{\rightsquigarrow} \cdots$, since the size of the programs are strictly decreasing with each passive step.

- For any P, the number of R and Q such that $P \overset{R}{\rightsquigarrow} Q$ is finite.

In fact, $\rightsquigarrow$ satisfies a "strong diamond property", namely that if $P \overset{R_1}{\rightsquigarrow} P_1$ and $P \overset{R_2}{\rightsquigarrow} P_2$ with $P_1 \neq P_2$, then there is a Q such that $P_1 \overset{R_2}{\rightsquigarrow} Q$ and $P_2 \overset{R_1}{\rightsquigarrow} Q$.

The graph form of a program P is rooted directed finite acyclic graph formed by (i) forming the passive-graph according to the $\rightsquigarrow$ relation, and then (ii) mapping the function $\lceil \_ \rceil$ over the nodes of the passive-graph to extract their active reductions. So, for example, taking the program $(\mathbf{r}_1 \,;\, \mathbf{r}_2) \# (\mathbf{r}_3 \,;\, \mathbf{r}_4)$ we (i) construct the passive graph:

$$\begin{tikzcd}
 & (\mathbf{r}_1 \,;\, \mathbf{r}_2) \# (\mathbf{r}_3 \,;\, \mathbf{r}_4) \arrow[dl, "\{\mathbf{r}_3\}"'] \arrow[dr, "\{\mathbf{r}_1\}"] & \\
(\mathbf{r}_1 \,;\, \mathbf{r}_2) \# \mathbf{r}_4 \arrow[dr, "\{\mathbf{r}_1\}"'] & & \mathbf{r}_2 \# (\mathbf{r}_3 \,;\, \mathbf{r}_4) \arrow[dl, "\{\mathbf{r}_3\}"] \\
 & \mathbf{r}_2 \# \mathbf{r}_4 &
\end{tikzcd}$$

and (ii) abstract the active region from each node to obtain:

$$\begin{tikzcd}
 & \{\mathbf{r}_1, \mathbf{r}_3\} \arrow[dl, "\{\mathbf{r}_3\}"'] \arrow[dr, "\{\mathbf{r}_1\}"] & \\
\{\mathbf{r}_1, \mathbf{r}_4\} \arrow[dr, "\{\mathbf{r}_1\}"'] & & \{\mathbf{r}_2, \mathbf{r}_3\} \arrow[dl, "\{\mathbf{r}_3\}"] \\
 & \{\mathbf{r}_2, \mathbf{r}_4\} &
\end{tikzcd}$$

3.2 Reasoning from Graphs

It should be clear from Proposition 8 that the transition traces of a program can be constructed from its graph. In fact, from the point of view of giving the operational semantics a program P, we can use the *tree* corresponding to the graph.

We will show how the graph representation can be used to reason about strong equivalence and strong approximation between programs. For the purpose of the behaviours (or transition traces) of programs, only need the set of complete paths through the graph.

Let $\mathbf{paths}(P)$ denote the complete (and necessarily finite) paths in the graph of P. So the graph of a program $\mathbf{r}_1 \,;\, (\mathbf{r}_2 \# \mathbf{r}_3)$ is just $\{\mathbf{r}_1\} \xrightarrow{\{\mathbf{r}_1\}} \{\mathbf{r}_2, \mathbf{r}_3\}$, and so the program has just a single path, $\langle \{\mathbf{r}_1\}\, \{\mathbf{r}_1\}\, \{\mathbf{r}_2, \mathbf{r}_3\} \rangle$

The domain of paths (ranged over by p_1, p_2 etc.) are the finite, nonempty odd-length sequences of sets of reduction relations. Writing concatenation of sequences by juxtaposition, if R, R_1, R_2 etc. range over sets of reduction relations, then a path is either a sequence of length one, $\langle R \rangle$, or a sequence of the form $\langle R_1, R_2 \rangle p$ for some path p. Alternatively we will denote a path by $\langle n_1 a_1 n_2 a_2 \ldots a_{k-1} n_k \rangle$ where the n_i (nodes) and a_i (arcs) are again sets of reduction relations.

The paths of a composed reduction system can be defined directly by induction on the passive steps:

Definition 9 *The paths of a program P,* $\mathbf{paths}(P)$*, is the least set of nonempty sequences of sets of reduction relations, such that:*

- *if* $P{\downarrow}$ *then* $[P] \in \mathbf{paths}(P)$
- *if* $P \stackrel{R}{\leadsto} P'$ *and* $p' \in \mathbf{paths}(P')$ *then* $\langle \lceil P \rceil, R \rangle p' \in \mathbf{paths}(P)$.

Now, in turn, the transition traces of a program can be defined in terms of its paths. This is given in the appendix.

The first implication of this is that if two programs have equivalent paths, then they must be strongly equivalent. We conjecture a tighter relationship, namely:

Conjecture 10 $\mathbf{paths}(P_1) = \mathbf{paths}(P_2)$ *if and only if* $(P_1 \equiv_o P_2)$ *is provable from the equational theory generated by the laws:*

(i) $\mathbf{r} \equiv_o \mathbf{r} \mathbin{+\!\!\!+} \mathbf{r}$ $\quad (ii)$ $\mathbin{+\!\!\!+}$ *is associative and commutative* $\quad (iii)$; *is associative.*

In fact, other than a few laws for the desynchroniser Δ, we have not found any other strong equivalences (than those derivable from the above). The inequational theory for $\sqsubseteq_o$ is, however, much richer. But proving inequalities from the transition traces (so far the only method we have) is rather tedious. Now we consider how to reason about $\sqsubseteq_o$ by building comparison relations on path-sets.

3.3 Path Comparisons

Each "node" represent the possible reductions possible at that node. The reductions on each "edge" represent the termination condition—a set of reductions which must be inapplicable for control to transfer along that edge. Comparing two paths of the same length, one path describes a broader range of behaviours than another, if it has at least as many reductions at each corresponding node (the odd elements of the sequence) but no more reductions on each edge (the even elements of the sequence). This leads to the following:

Definition 11 (Path Inclusion)
Two paths of equal length,
$p = \langle n_1, a_1, n_2 \ldots a_{k-1}, n_k \rangle$ *and* $p' = \langle n'_1, a'_1, n'_2 \ldots a'_{k-1}, n'_k \rangle$,
are in the path-inclusion ordering, written $p \leq p'$, *if*

1. $n_k = n'_k$,
2. $n_i \subseteq n'_i$, *for all* $i < k$, *and*
3. $a'_i \subseteq a_i$, *for all* $i \leq k$.

The path inclusion ordering is defined on composed reduction systems as:
$P \leq Q$ *if and only if for all* $p \in \mathbf{paths}(P)$ *there exists a path* $q \in \mathbf{paths}(Q)$ *such that* $p \leq q$.

Note that there is a stronger condition on the last node of a path. This is because the last node carries additional significance, since it is also a termination condition.

Proposition 12 $P \leq Q \Rightarrow P \sqsubseteq_o Q$

PROOF Since the transition traces are easily constructed from the paths, the proposition can be proved by showing that if $P \leq Q$ then the transition traces of P are contained in those of Q. Given the fact that the behaviours are extractable from the paths, a more direct (and arguably more useful) proof can be given by a compositional definition of the paths of a program. A compositional construction of paths is given in the appendix. $\Box$

Consider, for example, the composed reduction system $\mathbf{r}_1 ; \mathbf{r}_3 ; (\mathbf{r}_2 \mathbin{+\!\!\!+} \mathbf{r}_4)$:

$$\mathbf{paths}(\mathbf{r}_1 ; \mathbf{r}_3 ; (\mathbf{r}_2 \mathbin{+\!\!\!+} \mathbf{r}_4)) = \{\langle\{\mathbf{r}_1\}, \{\mathbf{r}_1\}\{\mathbf{r}_3\}\{\mathbf{r}_3\}\{\mathbf{r}_2, \mathbf{r}_4\}\rangle\}.$$

Since we have $\langle\{\mathbf{r}_1, \mathbf{r}_3\}, \{\mathbf{r}_1\}\{\mathbf{r}_2, \mathbf{r}_3\}\{\mathbf{r}_3\}\{\mathbf{r}_2, \mathbf{r}_4\}\rangle \in \mathbf{paths}((\mathbf{r}_1 ; \mathbf{r}_2) \mathbin{+\!\!\!+} (\mathbf{r}_3 ; \mathbf{r}_4))$ then we can conclude that $\mathbf{r}_1 ; \mathbf{r}_3 ; (\mathbf{r}_2 \mathbin{+\!\!\!+} \mathbf{r}_4) \sqsubseteq_o (\mathbf{r}_1 ; \mathbf{r}_2) \mathbin{+\!\!\!+} (\mathbf{r}_3 ; \mathbf{r}_4)$.

Path Stuttering

The main limitation of the path-inclusion ordering is that we can only compare paths of equal length. So, for example, we cannot prove the inequaility:

$$(\mathbf{r}_1 \mathbin{+\!\!\!+} \mathbf{r}_3) ; (\mathbf{r}_2 \mathbin{+\!\!\!+} \mathbf{r}_4) \sqsubseteq_o (\mathbf{r}_1 ; \mathbf{r}_2) \mathbin{+\!\!\!+} (\mathbf{r}_3 ; \mathbf{r}_4)$$

since the path of $(\mathbf{r}_1 \mathbin{+\!\!\!+} \mathbf{r}_3) ; (\mathbf{r}_2 \mathbin{+\!\!\!+} \mathbf{r}_4)$ (there is only one) is shorter than all the paths of $(\mathbf{r}_1 ; \mathbf{r}_2) \mathbin{+\!\!\!+} (\mathbf{r}_3 ; \mathbf{r}_4)$.

The solution is to define an analogue of closure under stuttering, at the level of paths. We do not literally add stuttering paths, but rather, paths which give rise to stuttering. Consider a path of the form $p_1\langle n, a\rangle p_2$. The arc a represents an internal termination step. Operationally, after this step is performed, we could offer some reductions from a, say n', and none will be applicable—and hence we can converge for all reductions in n'. Hence the path $p_1\langle n, a, n', n'\rangle p_2$ describes no more (but no fewer) behaviours than $p_1\langle n, a\rangle p_2$. This leads us to a definition of *path stuttering equivalence*

Definition 13 *Let path stuttering equivalence, $=_{\mathrm{s}}$, be the least equivalence relation on paths such that*

for all paths p_1, p_2 (p_1 possibly empty), and for all sets of reductions n, a, n' such that $n' \subseteq a \subseteq n$,

$$p_1\langle n, a\rangle p_2 =_{\mathrm{s}} p_1\langle n, a, n', n'\rangle p_2$$

For example, $\langle\{\mathbf{r}_1, \mathbf{r}_2\}\{\mathbf{r}_1, \mathbf{r}_2\}\{\mathbf{r}_3\}\rangle =_{\mathrm{s}} \langle\{\mathbf{r}_1, \mathbf{r}_2\}\{\mathbf{r}_1, \mathbf{r}_2\}\{\mathbf{r}_1\}\{\mathbf{r}_1\}\{\mathbf{r}_3\}\rangle$. With the development that follows, we will be able to conclude that

$$(\mathbf{r}_1 \mathbin{+\!\!\!+} \mathbf{r}_2) ; \mathbf{r}_3 \equiv_o (\mathbf{r}_1 \mathbin{+\!\!\!+} \mathbf{r}_2) ; \mathbf{r}_1 ; \mathbf{r}_3$$

Now we use path stuttering equivalence to coarsen the path inclusion ordering. As before we define a preorder on paths, and extend this to programs in the obvious way:

Definition 14 (Stuttered Path Inclusion) *Two paths, p and p', are in the stuttered path-inclusion ordering, written $p \leq_s p'$ if there exists p_1, p_2 such that*

$$p =_{\mathrm{s}} p_1 \leq p_2 =_{\mathrm{s}} p'.$$

On composed reduction systems we define $P \leq_s Q$ if and only if for all $p \in \mathbf{paths}(P)$ there exists a path $q \in \mathbf{paths}(Q)$ such that $p \leq_s q$.

Proposition 15 $P \leq_s Q \Rightarrow P \sqsubseteq_o Q$

PROOF (Outline) It is sufficient to show that $P \leq_s Q \Rightarrow \ddagger\mathsf{T}[\![P]\!] \subseteq \ddagger\mathsf{T}[\![Q]\!]$. The main step is to show that the closed ($\ddagger$) traces corresponding to a path $p_1\langle n, a\rangle p_2$ are equal to the closed traces of $p_1\langle n, a, n', n'\rangle p_2$, whenever $n' \subseteq a \subseteq n$. We omit the details. □

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.

[Bac89a] R. Back. A method for refining atomicity in parallel algorithms. In *PARLE '89, volume II*, number 365 in LNCS. Springer-Verlag, 1989.

[Bac89b] R. Back. Refinement calculus, part ii: Parallel and reactive programs. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, number 430 in LNCS. Springer-Verlag, 1989.

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

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

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

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

[dBKPR91] F. S. de Boer, J. N. Kok, C. Palamidessi, and J. J. M. M. Rutten. The failure of failures in a paradigm for asynchronous communication. In *CONCUR '91*, number 527 in Lecture Notes in Computer Science, pages 111–126. Springer-Verlag, 1991.

[Dij76] E. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[DJ89] N. Dershowitz and J-P. Jouannaud. *Rewrite Systems*, volume B, chapter 15. North-Holland, 1989.

[Hin69] R. Hindley. An abstract form of the Church-Rosser theorem. *J. Symbolic Logic*, 34(1), 1969.

[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.

[Klo92] J. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II. OUP, 1992.

[Lam89] L. Lamport. A simple approach to specifying concurrent systems. *C. ACM*, 31(1):32–45, January 1989.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *TCS*, 94, 1992.

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

[Par79] D. Park. On the semantics of fair parallelism. In *Abstract Software Specifications (1979 Copenhagen Winter School Proceedings)*, number 86 in Lecture Notes in Computer Science, pages 504–526. Springer-Verlag, 1979.

[Plo76] G. D. Plotkin. A powerdomain construction. *Siam J. Comput.*, 5(3):452–487, September 1976.

[Ros73] B. Rosen. Tree-manipulating systems and Church-Rosser theorems. *J. ACM*, 20(1):160–187, January 1973.

[San93a] D. Sands. A compositional semantics of combining forms for Gamma programs. In *International Conference on Formal Methods in Programming and Their Applications.* Springer-Verlag, 1993.

[San93b] 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.

[Sta74] J. Staples. Church-Rosser theorems for replacement systems. In *Algebra and Logic: papers from the Summer research institute of the Australian Mathematical Society*, number 450 in Lecture Notes in Mathematics. Springer-Verlag, 1974.

A Compositional Definition of Transition Traces

To give the compositional construction of transition traces we need to define the appropriate sequential and parallel composition operators over sets of traces.

Notation In what follows we will adopt the following notation. If S is a set, then S^* will denote the set of finite sequences of elements from S, S^+ will denote the finite non-empty sequences, and S^∞ will denote the infinite sequences. The power-set is denoted by $\wp(S)$.

Note in particular that for a reduction relation $\rightarrow_{\mathbf{r}}$, we will write $(\rightarrow_{\mathbf{r}})^*$ to denote the finite sequence of pairs contained in $\rightarrow_{\mathbf{r}}$, and *not* the transitive-reflexive closure of the relation.

Sequential Composition Sequential composition has an easy definition. We just take all concatenations of the atomic traces of the components. As is usual, if α and β are traces, then $\alpha\beta$ denotes their concatenation, which is just α when α is infinite. Define the following sequencing operation for trace sets:

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

End-synchronised Merge Not surprisingly, parallel composition is described with the use of a merging combinator which interleaves traces. The peculiarities of parallel composition are prominent in the definition. To define the transition traces of $P_1 \mathbin{+\!\!\!+} P_2$ 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, or synchronisation, at the point of their termination. To build up the picture, suppose α and β in $(\mathbf{U} \times \mathbf{U})^+$ are traces of some programs P_1 and P_2 respectively. The set of all interleavings of α and β which correspond to possible executions of $P_1 \mathbin{+\!\!\!+} P_2$ can be given inductively by:

$$(M, M) \sharp (N, N) = \begin{cases} \{(M, M)\} & if \quad M = N \\ \emptyset & otherwise \end{cases}$$

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

To generalise the definition to incorporate the infinite traces as well as the finite, we need to define the interleavings via a *maximal* fixed-point rather than a *minimal* fixed point as implicit in the above definition. There are many possible ways of presenting this construction. We choose an implicit definition of the required maximal fixed-point:

Definition 16 *A function* $\mathbf{m}$*, from pairs of nonempty traces to sets of nonempty traces, is an end-synchronised merger (ESM), if the following conditions are satisfied:*

1. *if* $(M, N) \in \mathbf{m}(\beta, \gamma)$ *then* $\beta = \gamma = (M, N)$*;*

2. *if* $\alpha \in \mathbf{m}(\beta,\gamma)$ *then* $(M,N)\alpha \in \mathbf{m}((M,N)\beta,\gamma)$
$\quad\& \ (M,N)\alpha \in \mathbf{m}(\gamma,(M,N)\beta)$

Definition 17 *The generalised end-synchronised merge is given by the pointwise union of all end-synchronised mergers:*

$$\alpha \sharp \beta = \cup \{\mathbf{m}(\alpha,\beta) \mid \mathbf{m} \text{ is an ESM}\}$$

Note that $\sharp$ is an ESM (this follows from the Knaster-Tarski fixed-point theorem), and therefore the largest ESM. The corresponding relation on sets of traces will provide the denotation of parallel composition:

$$T_1 \oplus T_2 = \{\gamma \mid \alpha \in T_1, \beta \in T_2, \gamma \in \alpha \sharp \beta\}$$

Definition 18 *The compositional atomic trace mapping* $\mathsf{T}_c[\![\_]\!] : \mathbf{P} \to \wp((\mathbf{U} \times \mathbf{U})^+ \cup (\mathbf{U} \times \mathbf{U})^\infty)$ *is given by induction on the syntax as:*

$$\begin{array}{rcl} \mathsf{T}_c[\![\mathbf{r}]\!] & = & \ddagger((\rightarrow_\mathbf{r})^* \odot \{(M,M) \mid M{\downarrow}^\mathbf{r}\}) \ \cup \ \ddagger((\rightarrow_\mathbf{r})^\infty) \\ \mathsf{T}_c[\![P_1 \,;\, P_2]\!] & = & \ddagger(\mathsf{T}_c[\![P_2]\!] \odot \mathsf{T}_c[\![P_1]\!]) \\ \mathsf{T}_c[\![P_1 \mathbin{+\!\!\!+} P_2]\!] & = & \ddagger(\mathsf{T}_c[\![P_1]\!] \oplus \mathsf{T}_c[\![P_2]\!]) \end{array}$$

Soundness It is straightforward to show that $\mathsf{T}_c[\![\_]\!]$ is sound with respect to behaviours of a program. The following lemma gives the basic operational correspondence:

Lemma 19 *1.* $\langle P, M\rangle \to M \Rightarrow (M,M) \in \mathsf{T}_c[\![P]\!]$

2. $\langle P, M\rangle \to \langle P', N\rangle \ \& \ \alpha \in \mathsf{T}_c[\![P']\!] \Longrightarrow (M,N)\alpha \in \mathsf{T}_c[\![P]\!]$

Proposition 20 $\mathsf{T}_c[\![P]\!] \subseteq \mathsf{T}_c[\![Q]\!] \Longrightarrow P \sqsubseteq_o Q$

PROOF From Lemma 19 it is easy to see that $\mathsf{T}_c[\![P]\!] \subseteq \mathsf{T}_c[\![Q]\!] \Rightarrow \mathcal{B}(P) \subseteq \mathcal{B}(Q)$. 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

$$\mathsf{T}_c[\![P]\!] \subseteq \mathsf{T}_c[\![Q]\!] \Rightarrow \forall \mathbf{C}.\, \mathsf{T}_c[\![\mathbf{C}[P_1]]\!] \subseteq \mathsf{T}_c[\![\mathbf{C}[P_2]]\!].$$

Putting these together we have

$$\begin{array}{rcl} \mathsf{T}_c[\![P]\!] \subseteq \mathsf{T}_c[\![Q]\!] & \Rightarrow & \forall \mathbf{C}.\, \mathcal{B}(\mathbf{C}[P_1]) \subseteq \mathcal{B}(\mathbf{C}[P_2]) \\ & \iff & P_1 \sqsubseteq_o P_2. \end{array}$$

□

B Path Constructions

Transition Traces from Paths The transition traces can be constructed from the paths as follows (overloading the mapping $\mathsf{T}[\![\_]\!]$):

$$\begin{array}{rcl} \mathsf{T}[\![\langle R\rangle]\!] & = & (T_R)^\infty \cup ((T_R)^* \odot \{(M,M) \mid \mathbf{r} \in R, M{\downarrow}^\mathbf{r}\}) \\ \mathsf{T}[\![\langle R, R'\rangle\sigma]\!] & = & (T_R)^\infty \cup ((T_R)^* \odot \{(M,M) \mid \mathbf{r} \in R', M{\downarrow}^\mathbf{r}\} \odot \mathsf{T}[\![\sigma]\!]) \\ & \text{where} & T_R = \{(M,N) \mid \mathbf{r} \in R, M {\rightarrow_\mathbf{r}} N\} \end{array}$$

Paths Constructed Compositionally

Proposition 21 *The following equations uniquely characterise the paths of a program:*

$$
\begin{array}{rcl}
\mathbf{paths}(\mathbf{r}) & = & \langle\{\mathbf{r}\}\rangle \\
\mathbf{paths}(P\,;Q) & = & \{\sigma_1\langle R\rangle\sigma_2 \mid \sigma_1 \in \mathbf{paths}(P), \sigma_2 \in \mathbf{paths}(Q), R = \mathit{last}(\sigma_1)\} \\
\mathbf{paths}(P \between Q) & = & \{\sigma_1 \otimes \sigma_2 \mid \sigma_1 \in \mathbf{paths}(P), \sigma_2 \in \mathbf{paths}(Q)\}
\end{array}
$$

where

$$
\begin{array}{rcl}
\langle R\rangle \otimes \langle R'\rangle & = & \langle R \cup R'\rangle \\
\langle R_1, R_2\rangle\sigma \otimes \langle R'\rangle & = & \langle R'\rangle \otimes \langle R_1 R_2\rangle\sigma \\
 & = & \{\langle (R_1 \cup R'), R_2\rangle\sigma' \mid \sigma' \in \langle R'\rangle \otimes \sigma\} \\
\langle R_1, R_2\rangle\sigma_1 \otimes \langle R_1', R_2'\rangle\sigma_1' & = & \{\langle (R_1 \cup R_1'), R_2\rangle\sigma \mid \sigma \in \sigma_1 \otimes \langle R_1', R_2'\rangle\sigma_1'\} \\
 & \cup & \{\langle (R_1 \cup R_1'), R_2'\rangle\sigma' \mid \sigma' \in \langle R_1, R_2\rangle\sigma_1 \otimes \sigma_1'\}
\end{array}
$$