

Gamma and the Logic of Transition Traces*

S. J. Gay

Department of Computer Science,
Royal Holloway, University of London,
Egham, Surrey, UK
simon@dcs.rhnc.ac.uk

C. L. Hankin

Department of Computing,
Imperial College of Science,
Technology and Medicine,
180 Queen's Gate, London, UK
clh@doc.ic.ac.uk

Abstract

Gamma is a language of conditional multiset rewrites, which can be seen either as a parallel programming language or as a specification language for parallel algorithms. Taking the latter view, we are interested in program refinement and formal techniques for reasoning about it.

In the present paper we apply Abramsky's framework of *domain theory in logical form*, to systematically develop a program logic for Gamma from a denotational semantics. Our semantics is a domain-theoretic reformulation of the *transition trace semantics*, which was defined for Gamma by Sands and based on earlier work by Brookes. We obtain a logic and proof system which is sound for our chosen notion of operational approximation or refinement and, as we show by means of an example, can be used to reason about program correctness. A further interesting point is that our techniques should apply not only to Gamma but to more general situations in which transition trace semantics can be used.

1 Introduction

The programming language Gamma was originally introduced by Banâtre and Le Métayer [5] and has been developed further by Hankin, Le Métayer and Sands [10, 11]. It allows algorithms to be expressed without introducing any unnecessary sequentiality (that is, sequentiality which is not required by the logic of the algorithm); indeed, enforcing additional sequentiality can be rather difficult. It is intended for use as a specification language for parallel computation—algorithms can be described in Gamma at a very high level, and then transformed into programs for particular parallel architectures. It can also be considered as a programming language in its own right, and its definition includes an operational semantics which can in principle be implemented directly. In practice, more information about the target architecture is needed before an efficient implementation can be realised; a number of researchers have produced implementations for a variety of architectures [4, 7, 14].

Computation in Gamma consists of the application of conditional rewriting rules to a multiset of data elements. Such a rule takes the form

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

in which the *reaction condition* R is a predicate, and the *action* A maps a collection of data elements to the multiset of elements which will replace them. An application of this rule consists of finding, if possible, elements x_1, \dots, x_n of the multiset such that $R(x_1, \dots, x_n)$ is true, and replacing them by the elements of $A(x_1, \dots, x_n)$. This process is repeated until it is no longer possible to find suitable x_1, \dots, x_n , at which point the computation ceases and the resulting multiset represents the answer. A single rewriting rule of this form, which may also be written $(A \Leftarrow R)$, is called a *basic reaction*. Compound programs can be built from basic reactions by means of sequential and parallel composition operators. In the parallel composition $P + Q$, reactions from either P or Q can be applied at any time, and the program terminates when a state is reached in which neither P nor Q can perform a rewrite. The sequential composition $Q \circ P$ behaves as P

* This research was supported by Esprit Basic Research Project 9102 (Coordination).

$$\begin{array}{c}
\frac{a_1, \dots, a_n \in M \quad R(a_1, \dots, a_n)}{\langle (A \leftarrow R), M \rangle \rightarrow \langle (A \leftarrow R), (M - \{a_1, \dots, a_n\}) \uplus A(a_1, \dots, a_n) \rangle} \\
\frac{\neg \exists a_1, \dots, a_n \in M. R(a_1, \dots, a_n)}{\langle (A \leftarrow R), M \rangle \rightarrow M} \\
\frac{\langle Q, M \rangle \rightarrow M}{\langle P \circ Q, M \rangle \rightarrow \langle P, M \rangle} \qquad \frac{\langle Q, M \rangle \rightarrow \langle Q', M' \rangle}{\langle P \circ Q, M \rangle \rightarrow \langle P \circ Q', M' \rangle} \\
\frac{\langle P, M \rangle \rightarrow \langle P', M' \rangle}{\langle P + Q, M \rangle \rightarrow \langle P' + Q, M' \rangle} \qquad \frac{\langle Q, M \rangle \rightarrow \langle Q', M' \rangle}{\langle P + Q, M \rangle \rightarrow \langle P + Q', M' \rangle} \\
\frac{\langle P, M \rangle \rightarrow M \quad \langle Q, M \rangle \rightarrow M}{\langle P + Q, M \rangle \rightarrow M}
\end{array}$$

Figure 1: Operational Semantics of Gamma.

until a terminal state for P is reached, and then behaves as Q . Many examples of programming using these operators can be found elsewhere [5, 8, 10, 11, 15].

The version of Gamma considered in this paper only makes use of these operators, so that programs are defined by the grammar

$$P ::= (A \leftarrow R) \mid P \circ P \mid P + P.$$

Other papers [10, 11] have defined combinators called *tropes* which capture common styles of basic reaction, but the present study concentrates on the pure language.

The feature of Gamma which makes it relevant to parallel computation is the local nature of the rewriting rules. If the multiset is viewed as a shared data space, several processors could potentially perform rewrites simultaneously in separate parts of the multiset. However, all of the existing work on Gamma has restricted attention to an operational semantics in which parallel composition corresponds to interleaving of actions rather than simultaneous actions, and the present paper continues in this tradition. This means that we are, in effect, working with a more general language of conditional state transitions; all of our results would apply equally to this more general situation.

The operational semantics of Gamma is defined in Figure 1. A nonterminal configuration $\langle P, M \rangle$ consists of a program P and a multiset M ; a terminal configuration is simply a multiset M . We write \rightarrow^* for the transitive closure of \rightarrow , and say that the configuration $\langle P, M \rangle$ *diverges*, written $\langle P, M \rangle \uparrow$, if there is an infinite sequence of transitions

$$\langle P, M \rangle \rightarrow \langle P_1, M_1 \rangle \rightarrow \langle P_2, M_2 \rangle \rightarrow \dots$$

If Gamma is considered as a specification language, the question of program refinement becomes interesting: a program which is very abstract and close to a specification, can be refined into a program which captures a more detailed description of an algorithm and is closer to an implementation. For example, it may be desirable to refine a program containing a great deal of parallel composition into one which contains more sequential composition. For this reason, previous work on Gamma has studied various operational approximation relations with the aim of formulating general rules for program refinement [10, 11, 16]. The approximation relation which we will consider in this paper is defined as follows. First define, for any program P ,

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

This leads to an approximation relation on programs (the \leq_R relation of Hankin *et al.* [10]) defined by

$$P \leq Q \iff \mathcal{B}(P) \subseteq \mathcal{B}(Q).$$

Equivalently, $P \leq Q$ if and only if $\forall M, N. \langle P, M \rangle \rightarrow^* N \Rightarrow \langle Q, M \rangle \rightarrow^* N$. This is used in the standard way to define an observational precongruence, by demanding approximation in all program contexts \mathbf{C} .

$$P \sqsubseteq_{\circ} Q \iff \forall \mathbf{C}. \mathbf{C}[P] \leq \mathbf{C}[Q].$$

The corresponding congruence is denoted by \equiv_o .

The purpose of the present paper is to study a formal system for reasoning about operational approximation and equivalence in Gamma—a program logic. This logic is derived systematically from a denotational semantics of Gamma, by means of Abramsky’s framework of *domain theory in logical form* [3]. The semantics in question is the transition trace semantics, defined for Gamma by Sands [16] and based on earlier work by Brookes [6]. In fact, we need to work with a domain-theoretic reformulation of this semantics. According to Abramsky’s theory, soundness of the logic follows from the fact that denotational approximation is sound for operational approximation. The question of full abstraction for the transition trace semantics of Gamma is still open; if the semantics were fully abstract then the corresponding program logic would be complete, but at the time of writing we are unable to make this deduction.

In a previous paper [9] we carried out a similar programme, starting from a resumption-style semantics [12] and obtaining essentially the *transition assertion logic* of Errington *et al.* [8]. That work suffers from two drawbacks: the semantics is definitely not fully abstract, which means that the logic cannot be complete for operational approximation; and the transition assertion logic itself describes the possible executions of programs at such a fine level of granularity that it is rather awkward to use. The present paper makes improvements on both fronts: the question of full abstraction is at least still open, and the logic permits higher-level and more abstract reasoning.

The remainder of the paper is organised as follows. Section 2 describes the transition trace semantics as previously defined [11]. Section 3 reviews the key points of domain theory in logical form, and the techniques arising from it which will be applied to Gamma. Section 4 defines the domain-theoretic version of the transition trace semantics. In Section 5 we apply the general theory to obtain a language of logical assertions from the domain of transition traces, and in Section 6 we derive the proof system for the logic. Section 7 presents a simple example of reasoning with the logic, and in Section 8 we summarise our results and consider the possibilities for further work.

2 The Transition Trace Semantics

The denotational semantics which has been defined for Gamma [10, 11, 16] is based on ideas of Brookes [6] relating to the semantics of shared-variable parallel languages. The meaning of a program P is a set of sequences of pairs of multisets:

$$(M_1, N_1)(M_2, N_2) \dots$$

representing possible sequences of state transitions. Both finite and infinite sequences are included, to cater for terminating and nonterminating computations. The presence of the pair (M_1, N_1) indicates that there is a possible transition $\langle P, M_1 \rangle \rightarrow \langle P', N_1 \rangle$; similarly there is a possible transition $\langle P', M_2 \rangle \rightarrow \langle P'', N_2 \rangle$. The key feature of this style of semantics is that in general the states N_1 and M_2 are different, to allow for the fact that in between two applications of a rewrite from P , another program which has been placed in parallel with P may perform a rewrite of its own.

For the transition trace semantics to be useful, it should not distinguish between behaviours which only differ in trivial ways. For Gamma, this means considering sets of traces T which are closed under the following operations. Here, α ranges over finite sequences of multiset pairs, and β ranges over finite and infinite sequences.

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

Absorption corresponds to the fact that a program may perform an arbitrary sequence of transitions without interference from the environment. Conversely, stuttering corresponds to the fact that there may be arbitrary interference from the environment without any visible actions being performed by the program.

Let $\ddagger T$ denote the left-stuttering and absorption closure (henceforth just closure) of a set T . We now define the function $\mathbb{T}[\cdot]$ which yields the finite and infinite sequences of pairs of multisets which can arise from a program. In the terminology of Hankin *et al.* [11] these are the *atomic* traces.

$$\mathbb{T}[P] = \{ (M_0, N_0)(M_1, N_1) \dots (M_k, N_k) \mid \langle P, M_0 \rangle \rightarrow \langle P_1, N_0 \rangle \&$$

$$\begin{aligned}
& \langle P_1, M_1 \rangle \rightarrow \langle P_1, N_1 \rangle \& \langle P_k, M_k \rangle \rightarrow N_k \} \\
\cup & \\
& \{ (M_0, N_0)(M_1, N_1) \dots (M_i, N_i) \dots \mid \\
& \langle P, M_0 \rangle \rightarrow \langle P_1, N_0 \rangle \& \langle P_i, M_i \rangle \rightarrow \langle P_{i+1}, N_i \rangle, i \geq 1 \}.
\end{aligned}$$

The *transition traces* are then given by the closure of the atomic traces: $\ddagger T[[P]]$. This is the semantic function which we will use. It can be defined compositionally as follows.

$$\begin{aligned}
\ddagger T[[A \leftarrow R]] &= \ddagger ((\text{mediators}_{(A \leftarrow R)})^* \ ; \ \text{terminals}_{(A \leftarrow R)} \cup \ddagger (\text{mediators}_{(A \leftarrow R)})^\omega) \\
\ddagger T[[P \circ Q]] &= \ddagger (\ddagger T[[Q]] \ ; \ \ddagger T[[P]]) \\
\ddagger T[[P + Q]] &= \ddagger (\ddagger T[[P]] \oplus \ddagger T[[Q]])
\end{aligned}$$

The sets $\text{mediators}_{(A \leftarrow R)}$ and $\text{terminals}_{(A \leftarrow R)}$ are defined by

$$\begin{aligned}
\text{mediators}_{(A \leftarrow R)} &= \{ (M, N) \mid \langle (A \leftarrow R), M \rangle \rightarrow \langle (A \leftarrow R), N \rangle \} \\
\text{terminals}_{(A \leftarrow R)} &= \{ (M, N) \mid \langle (A \leftarrow R), M \rangle \rightarrow N \}.
\end{aligned}$$

The operation $\ ; \$ on trace sets is defined by

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

The definition of \oplus is more complicated:

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

where

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

and an *end-synchronised merger* (ESM) is a function \mathbf{m} from pairs of non-empty traces to sets of non-empty traces such that

1. if $(M, N) \in \mathbf{m}(\beta, \gamma)$ then $\beta = \gamma = (M, N)$
2. if $\alpha \in \mathbf{m}(\beta, \gamma)$ then $(M, N)\alpha \in \mathbf{m}((M, N)\beta, \gamma)$ and $(M, N)\alpha \in \mathbf{m}(\gamma, (M, N)\beta)$.

Finally, the justification for this sequence of definitions is the following result [11].

Proposition 2.1 $\ddagger T[[P]] \subseteq \ddagger T[[Q]] \Rightarrow P \sqsubseteq_o Q$.

We do not know whether the implication can be reversed, i.e. whether the transition trace semantics is fully abstract. Although Brookes has established full abstraction for transition trace semantics of other parallel languages [6], his methods do not apply to Gamma. His proofs make essential use of the fact that for every state transition, there is a program which can perform that transition and no other. This is not true in the case of Gamma, because any multiset rewrite can also take place in the context of any larger multiset. More work is required to determine whether there is a different proof strategy which will apply to Gamma.

3 Domain Theory in Logical Form

Abramsky [1, 3] has developed a very general framework for connecting denotational semantics and program logic. This framework will be described briefly here, before going on to apply it to Gamma.

Consider a typed language with a denotational semantics expressed in terms of some variety of domain: perhaps Scott domains or SFP domains. This means that for each type σ there is a domain $D(\sigma)$, and for each typed term $t : \sigma$ there is a corresponding element $\llbracket t \rrbracket$ of $D(\sigma)$. The language may have several type constructors, such as products or function spaces, and for each one there is a matching domain construction. This situation is very standard. The new dimension introduced by Abramsky's work is the association of a

propositional theory $\mathcal{L}(\sigma)$ with each type σ . The formulae of $\mathcal{L}(\sigma)$ are the possible assertions about terms of type σ . Each theory has meets, joins and an ordering, giving it the structure of a distributive lattice. For each type constructor there is a construction on the propositional theories, and so this gives an alternative *logical* semantics of types. These constructions give extra formula constructions to the propositional theories. The logical view extends to the terms as well: for each type σ there is a satisfaction relation between terms of type σ and formulae in $\mathcal{L}(\sigma)$, and this satisfaction relation is axiomatised by a proof system. To make the framework as general as possible, Abramsky works with a typed metalanguage, which is a simply typed λ -calculus with additional term constructions for each type constructor. The propositional theories relate to this metalanguage, and the proof system axiomatises satisfaction of properties by terms of the metalanguage. For any particular computational situation, the general theory is applied by expressing the desired denotational semantics in terms of a translation into the metalanguage and then specialising the logic to construct formulae relating to the particular combinations of constructions which have been used.

Of course, the whole point of this theory is that there is a very strong connection between the denotational semantics and the program logic. In addition to the notion of property arising from the logic, there is a semantic notion of property: a compact open subset of a domain. The set of terms whose denotations lie in a given compact open set is interpreted as the set of terms satisfying the corresponding property. Motivation for this view of compact open sets as properties can be found elsewhere [3]; it comes primarily from notions of observability.

A similar shift of view on the logical side leads to consideration of the set of properties satisfied by a given term. Such a set X is closed under conjunction and implication (\leq) and inaccessible by joins: if $a \vee b \in X$ then either $a \in X$ or $b \in X$. This latter property expresses the fact that the logic is constructive. In terms of distributive lattices, such a set X is a prime filter.

Now for the connection between the denotational and logical views. For any domain D , the set of compact open sets ordered by inclusion forms a lattice $K\Omega(D)$. For any distributive lattice \mathcal{L} , the set of prime filters ordered by inclusion forms a domain $\text{Spec}(\mathcal{L})$. For each type σ , the domain $D(\sigma)$ and the theory $\mathcal{L}(\sigma)$ are related in this way: $\mathcal{L}(\sigma) = K\Omega(D(\sigma))$ and $D(\sigma) = \text{Spec}(\mathcal{L}(\sigma))$. This relationship is an instance of Stone duality.

This means that the set of all properties satisfied by a term corresponds to the denotation of the term, and this gives a logical characterisation of denotational equivalence. In fact, the logical characterisation operates at the level of the denotational order: $\llbracket t \rrbracket \sqsubseteq \llbracket u \rrbracket$ if and only if every property satisfied by t is satisfied by u . The full benefit of this correspondence is obtained when the denotational order coincides with an operational approximation relation, because then the logical system can be used to deduce facts about the operational behaviour of programs.

Two existing applications of domain theory in logical form serve to illustrate the potential benefits for Gamma. Abramsky [2] has shown that in the case of CCS, it is possible to start from a denotational description of processes and obtain (essentially) Hennessy-Milner logic. The standard characterisation of strong bisimulation in terms of satisfaction of HML formulae then follows from the general theory. Jensen [13] started with a non-standard denotational semantics of a typed functional language, corresponding to an abstract interpretation for strictness analysis, and obtained a logic which could be used to reason about strictness properties.

4 The Domain of Transition Trace Sets

We will now present a domain-theoretic reformulation of the transition trace semantics. For the remainder of the paper we will restrict attention to Gamma programs which operate on multisets of natural numbers; it will be clear that the theory can be generalised to multisets of any type.

The first requirement is a domain of sets of transition traces, that is, a domain whose elements are sets of finite and infinite sequences of pairs of multisets of natural numbers. This leads to the first problem—we do not know of a general construction of a domain of multisets over a given domain. Possible solutions include developing such a construction; using a representation of multisets in terms of sets, and working with a suitable powerdomain; and constructing a specific domain of multisets of natural numbers. In order to avoid being sidetracked into too much domain theory, or becoming embroiled in complex encodings, we adopt an alternative: simply observe that there is a countable set of finite multisets of natural numbers, assume some

fixed enumeration of that set, and replace multisets by naturals in our domain. When we consider basic reactions, we will assume that the reaction conditions and actions are given in terms of the enumeration, and the proof system for our logic will assume the existence of a proof system for satisfaction of reaction conditions. This rather abstract approach is very general—recalling the point made in Section 1, that with the interleaving semantics Gamma can be seen as a language of conditional state transitions rather than multiset rewrites, it is clear that all we are assuming is a countable state space. The development of our logic will concentrate on features intrinsic to Gamma rather than the particular state space.

We will use the flat domain \mathbb{N} of strict natural numbers, defined by $\mathbb{N} = \text{rect } t.\mathbf{0} \oplus t$, where $\mathbf{0} = \mathbf{1}_\perp$, $\mathbf{1}$ is the one-point domain, and \oplus is coalesced sum. Corresponding to this domain are the expected term constructions in the metalanguage: two introduction rules and an elimination rule.

$$\frac{}{0 : \mathbb{N}} \quad \frac{M : \mathbb{N}}{\text{succ}(M) : \mathbb{N}} \quad \frac{M : \mathbb{N} \quad N_1, N_2 : \nu}{\text{case } M \text{ of } 0 \Rightarrow N_1 ; \text{succ}(x) \Rightarrow N_2 \text{ end} : \nu}$$

We also need the domain of booleans, defined by $\mathbb{B} = \mathbf{0} \oplus \mathbf{0}$, with the following term constructions.

$$\frac{}{\text{true} : \mathbb{B}} \quad \frac{}{\text{false} : \mathbb{B}} \quad \frac{M : \mathbb{B} \quad N_1, N_2 : \nu}{\text{if } M \text{ then } N_1 \text{ else } N_2 \text{ end} : \nu}$$

The type constructor \times is available, with these term constructions.

$$\frac{M : \sigma \quad N : \tau}{(M, N) : \sigma \times \tau} \quad \frac{M : \sigma \times \tau \quad N : \nu}{\text{let } M \text{ be } (x, y).N \text{ end} : \nu}$$

We will be interested in the type $\mathbb{N} \times \mathbb{N}$, representing state transitions.

Because the transition trace semantics involves both finite and infinite sequences of transitions, we use a domain constructor for lazy lists. In general, $\text{List}(\sigma)$ is defined by $\text{List}(\sigma) = \text{rect } t.\mathbf{1} + (\sigma \times t)$, where $+$ is separated sum. There are three term constructions.

$$\frac{}{\text{nil} : \text{List}(\sigma)} \quad \frac{M : \sigma \quad N : \text{List}(\sigma)}{M :: N : \text{List}(\sigma)} \quad \frac{M : \sigma \quad N_1, N_2 : \nu}{\text{case } M \text{ of nil} \Rightarrow N_1 ; x :: y \Rightarrow N_2 \text{ end} : \nu}$$

Finally, we use the Hoare (lower) powerdomain to give sets of transition traces. We will need the following metalanguage term constructions for $\mathcal{P}_l(\sigma)$.

$$\frac{M : \sigma}{\{M\} : \mathcal{P}_l(\sigma)} \quad \frac{M, N : \mathcal{P}_l(\sigma)}{M \cup N : \mathcal{P}_l(\sigma)} \quad \frac{M : \mathcal{P}_l(\sigma) \quad N : \mathcal{P}_l(\tau)}{\text{over } M \text{ extend } x.N \text{ end} : \mathcal{P}_l(\tau)}$$

The domain of transition trace sets is $\mathbb{T} = \mathcal{P}_l(\text{List}(\mathbb{N} \times \mathbb{N}))$.

Following Abramsky [3] we interpret types as SFP domains, and metalanguage terms as functions on these domains. The behaviour of these functions is as expected; for clarity, we will make a few comments about the powerdomain and its associated terms.

An element of $\mathcal{P}_l(\sigma)$ is a non-empty, Scott-closed subset of σ , and the order is subset inclusion. Since we are interested in inclusion of transition trace sets, which implies operational approximation, the order in the Hoare powerdomain is the correct one for our purposes, as long as the condition of Scott-closure does not interfere.

Starting with a set of transition traces, it is necessary to take its Scott-closure in order to produce an element of the powerdomain. Because each transition trace is a total list, and the total lists are maximal in the domain of lazy lists, this amounts to taking the down-closure. The following easy result means that subset inclusion is still the desired ordering.

Lemma 4.1 *Let U, V be sets of maximal elements of a domain D . Then $U \subseteq V \iff \downarrow U \subseteq \downarrow V$.*

It can also be shown that down-closure commutes with stuttering- and absorption-closure, which means that applying down-closure at the outer level of the semantic definitions is equivalent to applying it at the inner levels; it can therefore be ignored when the metalanguage is used to define the semantics, because we can inductively assume that every set is down-closed.

When defining the meaning of the singleton and union operations, down-closure must be taken into account: $\llbracket \{M\} \rrbracket = \downarrow \{\llbracket M \rrbracket\}$. The set $\{\perp\}$, which will appear at certain points in the definition of the transition trace semantics, is the closest we can get to the empty set; \perp itself is defined by $\perp = \mu x.x$. Note that Abramsky uses $\{\cdot\}$ for the metalanguage singleton operation and \uplus for the union; we are using $\{\cdot\}$ and \cup to avoid confusion with multiset operations. The meaning of *over* M *extend* $x.N$ *end* is essentially $\bigcup \{N[a/x] \mid a \in M\}$.

Now that the metalanguage is available, defining the domain-theoretic semantics becomes a matter of programming the operations used in the original definitions. For sequential composition we need to extend the `append` function (defined as usual) to sets of lists. Note that the behaviour of `append` with an infinite first argument is to ignore the second argument, which is precisely what we need for the sequential composition of an infinite trace with a finite trace. For parallel composition we need the function `syncmerge`, which does end-synchronised merging. For a basic reaction ($A \leftarrow R$) we need to assume the existence of functions $\bar{R} : \mathbb{N} \rightarrow \mathbb{B}$, which tests the reaction condition, and $\bar{A} : \mathbb{N} \rightarrow \mathcal{P}_l(\mathbb{N})$, which returns the set of possible rewritten states. Then the functions `mediators` and `terminals` can be programmed, along with `omega` which computes all finite and infinite iterations of lists from a given set. For all the cases, `stutter` and `absorb` closure (functions `stutter` and `absorb`) are also needed; it can be shown that applying `stutter` and `absorb` separately is sufficient to give the `stutter` and `absorb` closure.

A selection of these functions are defined in Figure 2, using a familiar functional programming style. The final definition of the domain-theoretic transition trace semantics is

$$\begin{aligned} \llbracket (A \leftarrow R) \rrbracket &= \text{stutter} (\text{absorb} (\text{seq} (\text{omega} (\text{mediators} (\bar{A}, \bar{R})), \text{terminals} (\bar{R})))) \\ \llbracket Q \circ P \rrbracket &= \text{stutter} (\text{absorb} (\text{seq} (\llbracket P \rrbracket, \llbracket Q \rrbracket))) \\ \llbracket P + Q \rrbracket &= \text{stutter} (\text{absorb} (\text{par} (\llbracket P \rrbracket, \llbracket Q \rrbracket))). \end{aligned}$$

5 Extracting the Domain Logic

According to the framework of domain theory in logical form, each type σ generates a corresponding language of formulae $L(\sigma)$, which is then made into a logical theory $\mathcal{L}(\sigma)$ by imposing an order \leq , corresponding to implication, and subjecting it to certain axioms. We can now describe the steps leading to the definition of the theory $\mathcal{L}(\mathbb{T})$. Rather than present the fully general definitions of $\mathcal{L}(\sigma)$ for each type constructor [3], we specialise them to our situation.

First of all, there are formulae at each type corresponding to truth, falsity, conjunction and disjunction.

$$t, f \in L(\sigma) \quad \frac{\phi, \psi \in L(\sigma)}{\phi \wedge \psi, \phi \vee \psi \in L(\sigma)}$$

In $L(\mathbb{N})$ there are formulae corresponding to the properties of “being zero” and “being successor of something satisfying a given property”.

$$0 \in L(\mathbb{N}) \quad \frac{\phi \in L(\mathbb{N})}{\text{succ}(\phi) \in L(\mathbb{N})}$$

Note that we use, for example, `succ(0)` for both the metalanguage term denoting 1 and the formula representing the property of being equal to 1. Which meaning is intended should always be clear from the context.

In $L(\mathbb{N} \times \mathbb{N})$ there are formulae corresponding to properties of the form “being a pair whose components satisfy given properties”.

$$\frac{\phi, \psi \in L(\mathbb{N})}{\phi \times \psi \in L(\mathbb{N} \times \mathbb{N})}$$

Similarly, $L(\text{List}(\mathbb{N} \times \mathbb{N}))$ has formula constructions corresponding to the metalanguage term constructions.

$$\text{nil} \in L(\text{List}(\mathbb{N} \times \mathbb{N})) \quad \frac{\phi \in L(\mathbb{N} \times \mathbb{N}) \quad \psi \in L(\text{List}(\mathbb{N} \times \mathbb{N}))}{\phi :: \psi \in L(\text{List}(\mathbb{N} \times \mathbb{N}))}$$

```

stutter1 :  $\mathbb{N} \times \text{List}(\mathbb{N} \times \mathbb{N}) \rightarrow \mathcal{P}_I(\text{List}(\mathbb{N} \times \mathbb{N}))$ 

  stutter1 ( $n, xs$ ) =  $\{(n, n)\} \cup \text{stutter1}(\text{succ}(n), xs)$ 

stutter2 :  $\text{List}(\mathbb{N} \times \mathbb{N}) \rightarrow \mathcal{P}_I(\text{List}(\mathbb{N} \times \mathbb{N}))$ 

  stutter2  $xs = \text{case } xs \text{ of nil} \Rightarrow \{\text{nil}\};$ 
     $y :: ys \Rightarrow (\text{stutter1}(0, y :: ys)) \cup \text{over}(\text{stutter2 } ys) \text{ extend } z. \{y :: z\} \text{ end}$ 
  end

stutter :  $\mathcal{P}_I(\text{List}(\mathbb{N} \times \mathbb{N})) \rightarrow \mathcal{P}_I(\text{List}(\mathbb{N} \times \mathbb{N}))$ 

  stutter  $u = \text{over } u \text{ extend } x. \text{stutter2 } x \text{ end}$ 

absorb1 :  $(\mathbb{N} \times \mathbb{N}) \times \text{List}(\mathbb{N} \times \mathbb{N}) \rightarrow \mathcal{P}_I(\text{List}(\mathbb{N} \times \mathbb{N}))$ 

  absorb1 ( $(a, b), xs$ ) = case  $xs$  of  $\text{nil} \Rightarrow \{\perp\};$ 
     $(c, d) :: ys \Rightarrow \text{if equal}(b, c) \text{ then } \{(a, d) :: ys\} \cup \text{absorb1}((a, d), ys) \text{ else } \{\perp\} \text{ end}$ 
  end

absorb2 :  $\text{List}(\mathbb{N} \times \mathbb{N}) \rightarrow \mathcal{P}_I(\text{List}(\mathbb{N} \times \mathbb{N}))$ 

  absorb2  $xs = \text{case } xs \text{ of nil} \Rightarrow \{\text{nil}\};$ 
     $y :: ys \Rightarrow (\text{over}(\text{absorb2 } ys) \text{ extend } z. \{y :: z\} \text{ end}) \cup \text{absorb1}(y, ys)$ 
  end

absorb :  $\mathcal{P}_I(\text{List}(\mathbb{N} \times \mathbb{N})) \rightarrow \mathcal{P}_I(\text{List}(\mathbb{N} \times \mathbb{N}))$ 

  absorb  $u = \text{over } u \text{ extend } y. \{\text{absorb2 } y\} \text{ end}$ 

omega :  $\mathbb{T} \rightarrow \mathbb{T}$ 

  omega  $x = x \cup (\text{over } x \text{ extend } y. \text{over}(\text{omega } x) \text{ extend } z. \{y :: z\} \text{ end end}$ 

par :  $\mathbb{T} \times \mathbb{T} \rightarrow \mathbb{T}$ 

  par ( $x, y$ ) = over  $x$  extend  $u. \text{over } y \text{ extend } v. \text{syncmerge}(u, v) \text{ end end}$ 

seq :  $\mathbb{T} \times \mathbb{T} \rightarrow \mathbb{T}$ 

  seq ( $x, y$ ) = over  $x$  extend  $u. \text{over } y \text{ extend } v. \{\text{append}(u, v)\} \text{ end end}$ 

```

Figure 2: Metalanguage Definitions (selection)

The Hoare powerdomain introduces the *may* modality \diamond .

$$\frac{\phi \in L(\text{List}(\mathbb{N} \times \mathbb{N}))}{\diamond\phi \in L(\mathcal{P}_i(\text{List}(\mathbb{N} \times \mathbb{N})))}$$

If $\phi \in L(\text{List}(\mathbb{N} \times \mathbb{N}))$, i.e. ϕ is a property which a list could satisfy, then $\diamond\phi \in L(\mathcal{P}_i(\text{List}(\mathbb{N} \times \mathbb{N})))$ is a property which a set of lists could satisfy: namely, the property of containing at least one element satisfying ϕ .

In each case, more general formulae can be constructed by means of conjunction and disjunction.

For each σ there is a relation \leq on $L(\sigma)$, which is to be thought of as logical consequence. Various axioms are imposed, and $\mathcal{L}(\sigma)$ consists of equivalence classes of formulae under the equivalence defined by

$$\phi = \psi \iff \phi \leq \psi \text{ and } \psi \leq \phi.$$

The first group of axioms, present for every σ , give $\mathcal{L}(\sigma)$ a distributive lattice structure with t, f as top and bottom elements and \wedge, \vee as meet and join.

There are also various type-specific axioms, which describe how the distributive lattice structure interacts with formula constructors such as $\text{succ}(\cdot)$ and \diamond . They can be summarised as follows: each constructor is monotonic with respect to \leq , and each constructor commutes with arbitrary joins. For example, joins can be moved through the top-level \diamond :

$$\diamond(\phi :: (\bigvee_i \psi_i)) = \bigvee_i \diamond(\phi :: \psi_i) \quad \diamond((\bigvee_i \phi_i) :: \psi) = \bigvee_i \diamond(\phi_i :: \psi).$$

6 The Proof System

The general framework of domain theory in logical form includes, for each type σ , a proof system for a satisfaction relation between metalanguage terms of type σ and properties in $\mathcal{L}(\sigma)$. This means that the proof system for our Gamma logic can be derived systematically by specialising the general proof rules to the particular metalanguage terms used to define the semantics. Not surprisingly, the derivations are rather long and involved. An easier approach is to take a higher-level look at our metalanguage programs, formulate the proof rules which we expect to obtain, and then check that they really are valid. In the present paper we are omitting the detailed verifications, for reasons of space.

Because the transition trace sets are closed under stuttering and absorption, we obtain proof rules for stuttering and absorption, as follows.

$$\frac{P \vdash \diamond(\theta\psi)}{P \vdash \diamond(\theta(\phi, \phi)\psi)} \psi \neq \text{nil} \quad \frac{P \vdash \diamond(a(\phi, \psi)(\psi, \theta)b)}{P \vdash \diamond(a(\phi, \theta)b)}$$

To formulate the proof rules for parallel and sequential composition, it is convenient to define operations of parallel and sequential composition on formulae.

$$\frac{P \vdash \phi \quad Q \vdash \psi}{Q \circ P \vdash \psi \circ \phi} \quad \frac{P \vdash \phi \quad Q \vdash \psi}{P + Q \vdash \phi \parallel \psi}$$

To define the formula $\psi \circ \phi$, first assume without loss of generality that $\phi = \bigvee_i \diamond\phi_i$ and $\psi = \bigvee_j \diamond\psi_j$ with each ϕ_i, ψ_j a list of pairs of formulae. Then

$$\psi \circ \phi = \bigvee_{i,j} \diamond\psi_j \circ \phi_i$$

where $\psi_j \circ \phi_i$ is defined by appending lists of pairs.

With the same assumption,

$$\phi \parallel \psi = \bigvee_{i,j} \bigwedge_{\theta} \diamond\theta$$

where θ ranges over end-synchronised merges of ϕ and ψ .

For a basic reaction there are two rules, the first corresponding to the terminals part of the semantic definition, and the second allowing any number of steps corresponding to mediators to be added.

$$\frac{\phi \Rightarrow \neg R}{(A \Leftarrow R) \vdash \diamond(\phi, \phi)} \quad \frac{\phi \Rightarrow R \quad A(\phi) \Rightarrow \psi \quad (A \Leftarrow R) \vdash \diamond\theta}{(A \Leftarrow R) \vdash \diamond((\phi, \psi)\theta)}$$

The hypothesis $\phi \Rightarrow \neg R$ means

$$\forall M.M \vdash \phi \Rightarrow \neg \exists \bar{x} \subseteq M.R(\bar{x}),$$

where M ranges over multisets. The combined hypothesis $\phi \Rightarrow R, A(\phi) \Rightarrow \psi$ means

$$\forall M.[M \vdash \phi \Rightarrow \exists \bar{x} \subseteq M.[R(\bar{x}) \wedge M[A(\bar{x})/\bar{x}] \vdash \psi]].$$

We need to assume the existence of a proof system for establishing such assertions. Given a sound and complete proof system for these assertions, Abramsky's general results on domain logic yield the following.

Proposition 6.1 $\forall \phi.[P \vdash \phi \Rightarrow Q \vdash \phi] \iff \ddagger T[[P]] \subseteq \ddagger T[[Q]]$.

Combined with Proposition 2.1, we obtain:

Proposition 6.2 $\forall \phi.[P \vdash \phi \Rightarrow Q \vdash \phi] \Rightarrow P \sqsubseteq_o Q$.

If the transition trace semantics could be shown to be fully abstract, this implication could be reversed. Determining whether or not the semantics is fully abstract is a subject for future work.

7 Example

We will now demonstrate the use of our program logic by proving something about a Gamma program. The example we consider is the Gamma program for calculating Fibonacci numbers [8].

If we define

$$\begin{aligned} P_1 &= x \rightarrow \{x-1, x-2\} \Leftarrow (x > 1) \\ P_2 &= x \rightarrow \{1\} \Leftarrow (x = 0) \\ P_3 &= x, y \rightarrow \{x+y\} \Leftarrow true \end{aligned}$$

then the program $P = P_3 \circ (P_1 + P_2)$ calculates Fibonacci numbers: when applied to a multiset containing the single number n_0 it eventually terminates with a multiset containing just the n_0 th Fibonacci number, f_{n_0} (where $f_0 = 1$). The aim of our example is to prove an instance of this general fact: that when given the multiset $\{4\}$, P can terminate with the multiset $\{5 = f_4\}$. Formally, we wish to derive

$$P_3 \circ (P_1 + P_2) \vdash \diamond(\{4\}, \{5\}).$$

First consider P_1 . Using the terminal rule, we have

$$P_1 \vdash \diamond(\{1, 1, 1, 1, 1\}, \{1, 1, 1, 1, 1\})$$

and by repeated applications of the mediator rule, we can deduce

$$\begin{aligned} P_1 \vdash & \diamond(\{4\}, \{3, 2\})(\{3, 2\}, \{2, 2, 1\})(\{2, 2, 1\}, \{2, 1, 1, 0\}) \\ & (\{2, 1, 1, 0\}, \{1, 1, 1, 0, 0\})(\{1, 1, 1, 1, 1\}, \{1, 1, 1, 1, 1\}). \end{aligned}$$

Notice the jump from $\{1, 1, 1, 0, 0\}$ to $\{1, 1, 1, 1, 1\}$, corresponding to a rewrite which will be performed by P_2 . Using absorption we can hide some of the intermediate steps, obtaining

$$P_1 \vdash \diamond(\{4\}, \{1, 1, 1, 0, 0\})(\{1, 1, 1, 1, 1\}, \{1, 1, 1, 1, 1\}).$$

For P_2 , the terminal rule gives

$$P_2 \vdash \diamond(\{1, 1, 1, 1, 1\}, \{1, 1, 1, 1, 1\})$$

which can be extended by repeated application of the mediator rule to

$$P_2 \vdash \diamond(\{1, 1, 1, 0, 0\}, \{1, 1, 1, 1, 0\})(\{1, 1, 1, 1, 0\}, \{1, 1, 1, 1, 1\})(\{1, 1, 1, 1, 1\}, \{1, 1, 1, 1, 1\}).$$

Again, absorption allows intermediate steps to be hidden. Notice that we keep the final repeated state, which represents termination; this is required by the definition of end-synchronised merge.

$$P_2 \vdash \diamond(\{1, 1, 1, 0, 0\}, \{1, 1, 1, 1, 1\})(\{1, 1, 1, 1, 1\}, \{1, 1, 1, 1, 1\}).$$

Using the parallel rule, we can derive

$$P_1 + P_2 \vdash \diamond(\{4\}, \{1, 1, 1, 0, 0\})(\{1, 1, 1, 0, 0\}, \{1, 1, 1, 1, 1\})(\{1, 1, 1, 1, 1\}, \{1, 1, 1, 1, 1\})$$

which, by absorption, reduces to

$$P_1 + P_2 \vdash \diamond(\{4\}, \{1, 1, 1, 1, 1\}).$$

Similarly, for P_3 we can derive

$$P_3 \vdash \diamond(\{1, 1, 1, 1, 1\}, \{5\}).$$

Finally, the sequential rule gives

$$P_3 \circ (P_1 + P_2) \vdash \diamond(\{4\}, \{1, 1, 1, 1, 1\})(\{1, 1, 1, 1, 1\}, \{5\})$$

and hence, by more absorption,

$$P_3 \circ (P_1 + P_2) \vdash \diamond(\{4\}, \{5\})$$

as required.

8 Conclusions

Starting with a domain-theoretic reformulation of the transition trace semantics of Gamma, we have applied Abramsky's framework of domain theory in logical form to obtain a program logic. This logic is sound for our chosen notion of operational approximation or refinement, for quite general reasons. Our only assumption is the existence of a proof system for establishing appropriate properties of multisets, in particular the existence or non-existence of collections of elements satisfying given reaction conditions.

It would be very satisfying to eliminate this assumption, by deriving a logic of multisets in the same systematic way. This would require a general domain constructor for multisets; if such a constructor could be developed, we would be able to modify our domain of transition traces to include the details of the multiset level. However, there is another potential difficulty: in the proof rule which establishes the termination of a basic reaction, there is a negative hypothesis (the non-existence of a reacting collection of elements). This negative hypothesis appears to be at odds with the generally intuitionistic nature of domain logic; it remains to be seen whether or not it would be a genuine problem. In any case, our present technique of working with an enumeration of the set of finite multisets seems to yield a convenient level of generality; it also means that we may be able to adapt our logic to other languages which have been given transition trace semantics.

Another open question is that of full abstraction of the transition trace semantics. In situations where the semantics is fully abstract, domain theory in logical form produces stronger results, i.e. the exact logical characterisation of operational approximation. More work is needed in order to settle the full abstraction question for Gamma.

Our results are interesting for two reasons. First, they provide a program logic for Gamma, which was our aim. This logic is easier to use than previous proposals [8, 9] as its description of Gamma computations is higher-level. Second, we have added to the pool of applications of Abramsky's general theory. In comparison with previous applications (to the lazy λ -calculus [1] and synchronisation trees with strong bisimulation [2]) our domain $\mathbb{T} = \mathcal{P}_i(\text{List}(\mathbb{N} \times \mathbb{N}))$ has a different character. It is not a simple recursively-defined domain, and the semantic definitions (incorporating stuttering- and absorption-closure) are rather more complicated.

We have two main applications in mind for the results of the present paper. One is simply that once reasoning about Gamma programs has been formalised, it becomes possible to investigate automating the verification process. Another interesting possibility is inspired by Jensen's work on strictness logic [13]. He applied Abramsky's theory to a non-standard semantics of a functional language, using abstract domains for strictness analysis, and obtained a formal system for reasoning about strictness properties. Adapting this idea to Gamma is a potential area of future work.

References

- [1] S. Abramsky. *Domain Theory and the Logic of Observable Properties*. PhD thesis, University of London, October 1987.
- [2] S. Abramsky. A domain equation for bisimulation. *Information and Computation*, 92(2):161–218, June 1991.
- [3] S. Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 51:1–77, 1991.
- [4] J.-P. Banâtre, A. Coutant, and D. Le Métayer. Parallel machines for multiset transformation and their programming style. *Informationstechnik*, 30:99–109, 1988.
- [5] J.-P. Banâtre and D. Le Métayer. The GAMMA model and its discipline of programming. *Science of Computer Programming*, 15:55–77, 1990.
- [6] S. D. Brookes. Full abstraction for a shared variable parallel language. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 98–109. IEEE Computer Society Press, 1993.
- [7] C. Creveuil. Implementation of Gamma on the connection machine. In D. Le Métayer, editor, *Proceedings of the Workshop on Research Directions in High-Level Parallel Programming Languages*, volume 574 of *Lecture Notes in Computer Science*, pages 219–230. Springer-Verlag, 1991.
- [8] L. Errington, C. L. Hankin, and T. P. Jensen. Reasoning about Gamma programs. In G. L. Burn, S. J. Gay, and M. D. Ryan, editors, *Theory and Formal Methods 1993: Proceedings of the First Imperial College Department of Computing Workshop on Theory and Formal Methods*, Workshops in Computing. Springer-Verlag, 1993.
- [9] S. J. Gay and C. L. Hankin. A program logic for Gamma. In *Coordination Models and Languages*. Imperial College Press, 1996. To appear.
- [10] C. L. Hankin, D. Le Métayer, and D. Sands. A calculus of Gamma programs. Technical Report 672, INRIA-RENNES, October 1992.
- [11] C. L. Hankin, D. Le Métayer, and D. Sands. The coordination language Gamma: Semantics and transformation. Submitted for publication, 1995.
- [12] M. Hennessy and G. Plotkin. Full abstraction for a simple parallel programming language. In J. Beçvar, editor, *Mathematical Foundations of Computer Science*, Berlin, 1979. Springer-Verlag. Lecture Notes in Computer Science Vol. 74.
- [13] T. P. Jensen. Strictness analysis in logical form. In *FPCA91*, 1991.
- [14] H. Kuchen and K. Gladitz. Implementing bags on a shared memory MIMD machine. In *Proceedings of the 4th International Workshop on Parallel Implementation of Functional Languages*, 1992. Also University of Aachen Technical Report TR92-19.
- [15] H. McEvoy. Gamma, chromatic typing and vegetation. Technical report, University of Amsterdam, 1994.
- [16] D. Sands. Laws of parallel synchronised termination. In G. L. Burn, S. J. Gay, and M. D. Ryan, editors, *Theory and Formal Methods 1993: Proceedings of the First Imperial College Department of Computing Workshop on Theory and Formal Methods*, Workshops in Computing. Springer-Verlag, 1993.