

A PROGRAM LOGIC FOR GAMMA

S. J. GAY

*Department of Computer Science,
Royal Holloway, University of London,
Egham, Surrey TW20 0EX, UK*

C. L. HANKIN

*Department of Computing,
Imperial College of Science, Technology and Medicine,
180 Queen's Gate, London SW7 2BZ, UK*

We take a systematic approach to the construction of a program logic for Gamma, by applying Abramsky's domain theory in logical form to a denotational semantics of the language. Starting from a resumption semantics of Gamma, we are able to derive both the formulae and the proof system of the transition assertion logic previously proposed by Errington, Hankin and Jensen. The general theory enables us to prove soundness of the logic, although completeness fails because the resumption semantics of Gamma is not fully abstract. At the end of the paper we discuss the possibilities for obtaining a complete logic.

1 Introduction

The programming language Gamma was originally introduced by Banâtre and Le Métayer⁴ and has been developed further by Hankin, Le Métayer and Sands.^{8,9} 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, although in practice more information about the target architecture is needed before an efficient implementation can be realised.

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 is a multiset of data elements. 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 $P \circ Q$ behaves as P until a terminal state for P is reached, and then behaves as Q . Many examples of programming using these operators can be found elsewhere.^{4,7,8,9}

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^{8,9} 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.

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

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

As an illustration of the Gamma programming style, consider the following program for calculating Fibonacci numbers. It consists of three subprograms:

$$\begin{aligned} C_1 &= x \rightarrow \{x-1, x-2\} \Leftarrow (x > 1) \\ C_2 &= x \rightarrow \{1\} \Leftarrow (x = 0) \\ C_3 &= x, y \rightarrow \{x+y\} \Leftarrow \text{true} \end{aligned}$$

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

Figure 1: Operational Semantics of Gamma.

which are combined in to the program $C = C_3 \circ (C_1 + C_2)$. When applied to a multiset containing the single number n , C eventually terminates with a multiset containing just f_n , the n th Fibonacci number (where $f_0 = 1$). The programs C_1 and C_2 between them replace the original number n by f_n 1s, which are then added together by C_3 . At first sight, this algorithm is not very intuitive; however, we will return to it in Section 7 and give a proof of its correctness.

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 lot 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.^{8,9,15}

The purpose of the present paper is to study a formal system for reasoning about operational approximation and equivalence in Gamma—a program logic. The logic is almost the same as the existing *transition assertion logic* for Gamma,⁷ but now we use Abramsky’s framework of *domain theory in logical form*^{1,3} to derive the logic systematically from a denotational semantics. Soundness of the logic follows from the general theory, although we do not ob-

tain completeness because the denotational semantics fails to be fully abstract. This point will be discussed in more detail later.

The remainder of the paper is organised as follows. Section 2 describes the transition assertion logic as previously defined,⁷ and its proof system. 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 a denotational semantics of Gamma, using a domain of resumptions.^{11,14} In Section 5 we apply the general theory to obtain a logic from the resumption domain, and see that the language of transition assertions is recovered in a slightly modified form. Section 6 describes how we obtain the proof rules of the transition assertion logic, and Section 7 reworks an existing example of program verification with the modified logic. Section 8 discusses the issues involved in obtaining a sound and complete logic, and in Section 9 we consider the possibilities for extensions and improvements of our results.

2 The Transition Assertion Logic

The existing program logic for Gamma is the *transition assertion logic*.⁷ It is based on Brookes' logic for a shared-variable parallel language.⁵

Transition assertions have the following syntax.

$$\begin{array}{l} \phi ::= P \sum_{i \in I} P_i \phi_i \quad \textit{branching} \\ | P \bullet \quad \textit{terminal} \\ | \theta \quad \textit{variable} \\ | \mu \theta. \phi \quad \textit{recursion} \\ | \phi \wedge \phi \quad \textit{conjunction} \end{array}$$

In this definition, P ranges over properties of multisets. In the original formulation of the transition assertion logic, each summand in a branching assertion was associated with a label describing a multiset substitution. These labels are omitted from the discussion in the present paper; they could be recovered by comparing the multisets before and after a transition. The fixpoint operator μ is a binding operator; the free and bound variables of an assertion can be defined in the usual way, and an assertion is said to be *closed* if it has no free variables. In the proof system which will be defined in a moment, $C \text{ sat } \phi$ indicates that the program C satisfies the closed assertion ϕ .

Writing $M \models P$ for satisfaction of the property P by the multiset M , we can define the meaning of the transition assertions as follows.

$$C \text{ sat } P \bullet \iff (M \models P) \Rightarrow \langle C, M \rangle \rightarrow M$$

$C \text{ sat } P \sum_{i \in I} P_i \phi_i$ is true if and only if

1. $(M \models P) \wedge \langle C, M \rangle \rightarrow \langle C', M' \rangle \Rightarrow \exists i \leq n. (M' \models P_i) \wedge (C' \mathbf{sat} \phi_i)$
2. $(M \models P) \Rightarrow \forall i \leq n. \langle C, M \rangle \rightarrow \langle C_i, M_i \rangle$

In other words, a branching assertion describes all of the possible transitions for the program C in a multiset satisfying P .

$$C \mathbf{sat} (\phi \wedge \psi) \iff (C \mathbf{sat} \phi) \wedge (C \mathbf{sat} \psi)$$

The meaning of a recursive assertion $\mu\theta.\phi$ is defined in terms of the finite unfoldings ϕ_i .

$$\begin{aligned} \phi_0 &= \mathbf{false} \bullet \\ \phi_{n+1} &= \phi[\phi_n/\theta] \end{aligned}$$

Then

$$C \mathbf{sat} \mu\theta.\phi \iff \forall n \geq 0. (C \mathbf{sat} \phi_n).$$

Formulating the proof system for the transition assertion logic requires some additional definitions. We need to assume that it is possible to prove assertions of the form $\{P\}A(\bar{x})/\bar{x}\{Q\}$, which means that if the multiset M satisfies P then the multiset obtained by substituting the multiset $A(\bar{x})$ for the tuple \bar{x} in M satisfies Q . The proof rule for a basic reaction is

$$\frac{\{P\}A(\bar{x})/\bar{x}\{P\}}{(A \Leftarrow R) \mathbf{sat} \mu\theta. \left[\begin{array}{l} (P \wedge R(\bar{x})) \sum (P \wedge \exists \bar{y}. \bar{y} = A(\bar{x}))\theta \\ \wedge (P \wedge \neg \exists \bar{x}. R(\bar{x}))\bullet \end{array} \right]}$$

The hypothesis is that P is an invariant for the action A . The recursive assertion in the conclusion corresponds to the repeated application of the rewrite. If the multiset satisfies $\neg \exists \bar{x}. R(\bar{x})$ then the program $(A \Leftarrow R)$ terminates. Otherwise, the branching assertion (in which the summation is over all possible ways of finding a tuple \bar{x} in the multiset such that $R(\bar{x})$ is true) describes the possible transitions, in each case adding the assertion that the multiset now contains a tuple \bar{y} which resulted from applying A to \bar{x} .

There is a straightforward rule for conjunction.

$$\frac{C \mathbf{sat} \phi \quad C \mathbf{sat} \psi}{C \mathbf{sat} \phi \wedge \psi}$$

Operations of sequential and parallel composition are defined on transition assertions, in such a way as to make the following proof rules valid.

$$\frac{C_1 \mathbf{sat} \phi \quad C_2 \mathbf{sat} \psi}{C_1 + C_2 \mathbf{sat} \phi \parallel \psi} \qquad \frac{C_1 \mathbf{sat} \phi \quad C_2 \mathbf{sat} \psi}{C_2 \circ C_1 \mathbf{sat} \psi \circ \phi}$$

The definitions of $\psi \circ \phi$ and $\phi \parallel \psi$, slightly modified for our purposes, are given in Section 6. For this reason, we will not reproduce the original definitions here.

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, such as 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 construction. 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 and 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. Abramsky² 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 (HML).¹⁰ The standard characterisation of strong bisimulation in terms of satisfaction of HML formulae then follows from the general theory. Jensen¹³ 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.

Zhang¹⁶ has described how the ideas of domain theory in logical form can be used to derive the formulae of Brookes' original transition assertion logic from a resumption-style semantics^{11,14} of the parallel language. In the next section, we define a resumption semantics of Gamma and extract the transition assertion logic from it. In fact we go further than Zhang: we not only recover the assertions, but also the proof system, with a slight modification to accommodate the fact that Abramsky's theory does not support recursively-defined formulae.

4 The Resumption Semantics of Gamma

Consider a language whose operational semantics is defined in terms of transitions of (program, state) pairs—for example Gamma, with the operational semantics defined in Section 1. The idea behind resumption semantics is to view a program as a function which maps a state to a pair consisting of a new state and a new program. This new program is the *resumption*, so called because the new state is considered to mark a pause after which execution resumes. The resumption style of semantics is typically used when the language has a parallel composition operator which allows programs to interfere with each other by modification of some shared state. The semantics of parallel composition is defined by interleaving, and the resumptions mark the points at which execution can switch between parallel programs. Thus any computation which takes place during a single transition is an atomic action which cannot be interrupted, and the degree of atomicity can be controlled by suitable definition of the transition relation.

The domain of resumptions for Gamma is defined by

$$\rho = \text{rec } t. (\mathcal{M}(\sigma) \rightarrow \mathcal{P}((\mathcal{M}(\sigma) \times t) + \mathcal{M}(\sigma))).$$

Here σ is a domain of the elements which exist in multisets, and $\mathcal{P}()$ is the Plotkin powerdomain. We will write $\{\cdot\}_P$ and \cup_P for the operations of singleton and union in the Plotkin powerdomain. Note that

$$\begin{aligned} \{x\}_P &= \{x\} \\ x \cup_P y &= \text{Con}(x \cup y) \end{aligned}$$

where Con is the convex closure operator.

Ideally $\mathcal{M}(\cdot)$ should be a general domain constructor for multisets, but as far as we know, no such construction has been described in the literature. Therefore we will assume that σ is a flat domain, such as int , and take $\mathcal{M}(\sigma)$ to be a flat domain consisting of multisets of non-bottom elements of σ .

We will define the resumption semantics of Gamma in terms of Abramsky's metalanguage for denotational semantics.³ This will allow us to demonstrate the application of the general theory in deriving the proof system for the logic.

The metalanguage needs to be extended with term constructions corre-

sponding to the domain of multisets.

$$\frac{A : \mathcal{M}(\sigma) \quad M : \sigma^n \quad N : \mathcal{M}(\sigma)}{\text{substitute } A \text{ for } M \text{ in } N : \mathcal{M}(\sigma)}$$

$$\frac{M : \mathcal{M}(\sigma) \quad R : \sigma^n \rightarrow \text{bool}}{\text{test } M \text{ for } R : \text{bool}}$$

$$\frac{M : \mathcal{M}(\sigma) \quad R : \sigma^n \rightarrow \text{bool}}{\text{extract } R \text{ from } M : \mathcal{P}(\sigma^n)}$$

The meaning of substitute A for M in N is the multiset obtained by removing the n elements of the tuple M from the multiset N and replacing them with the elements of the multiset A . The meaning of test M for R is true or false depending on whether or not the multiset M contains a tuple of elements satisfying the condition R . The meaning of extract R from M is the set of all tuples of elements from the multiset M which satisfy the condition R .

The rest of the metalanguage is standard, but it may be useful to explain one of the terms corresponding to the Plotkin powerdomain. Its introduction rule is

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

and the meaning of over M extend $\{x\}.N$ is $\uparrow(X) \cap \overline{X}$ where $X = \bigcup\{(\lambda x.N)y \mid y \in M\}$, $\uparrow(X)$ is the up-closure of X , and \overline{X} is the Scott-closure of X .

Now we can define the resumption semantics of Gamma.

$$\begin{aligned} \llbracket (A \Leftarrow R) \rrbracket = & \\ & \mu f. \lambda s. \text{ if } (\text{test } s \text{ for } \llbracket R \rrbracket) \text{ then} \\ & \quad \text{over } (\text{extract } \llbracket R \rrbracket \text{ from } s) \text{ extend } \{t\}. \\ & \quad \quad \{\text{inl}((\text{substitute } \llbracket A \rrbracket t \text{ for } t \text{ in } s, f))\} \\ & \text{ else} \\ & \quad \{\text{inr}(s)\} \end{aligned}$$

Given $f, g \in \rho$ we define $f ; g, f + g \in \rho$ as the least continuous functions such that

$$\begin{aligned} f ; g = & \\ & \lambda s. \text{ over } fs \text{ extend } \{x\}. \\ & \quad \text{case } x \text{ of} \\ & \quad \quad \text{inl}((t, h)) \Rightarrow \{\text{inl}((t, h ; g))\}; \\ & \quad \quad \text{inr}(t) \Rightarrow gs \\ & \text{ end} \end{aligned}$$

```

f + g =
  λs. over fs extend {x}.
    case x of
      inl((t, h)) ⇒
        over gs extend {y}.
          case y of
            inl((u, k)) ⇒ {inl((t, h + g))} ∪ {inl((u, f + k))};
            inr(u) ⇒ {inl((t, h + g))}
          end;
      inr(t) ⇒
        over gs extend {y}.
          case y of
            inl((u, k)) ⇒ {inl((u, f + k))};
            inr(u) ⇒ {inr(u)}
          end;
    end

```

and then $\llbracket Q \circ P \rrbracket = \llbracket P \rrbracket ; \llbracket Q \rrbracket$ and $\llbracket P + Q \rrbracket = \llbracket P \rrbracket + \llbracket Q \rrbracket$.

It is now possible to compare the resumption semantics and the operational semantics. The operational approximation relation which has been studied previously⁸ is defined as follows. First define, for any program P , a function $\mathcal{B}(P)$ by

$$\mathcal{B}(P)M = \{N \mid \langle P, M \rangle \rightarrow^* N\}.$$

This leads to an approximation relation on programs 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} .

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

The corresponding congruence is denoted by \equiv_o .

There is an alternative notion of behaviour, which takes divergence into account. For each program P and multiset M , define $\mathcal{B}'(P)M \in \mathcal{P}(\mathcal{M}(\sigma))$ by

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

Note that because $\mathcal{M}(\sigma)$ is a flat domain, the convexity condition in the definition of \cup_P becomes vacuous and we can simply interpret the definition of $\mathcal{B}'(P)M$ in terms of sets.

We can now define another approximation relation by

$$P \leq' Q \iff \mathcal{B}'(P) \sqsubseteq \mathcal{B}'(Q)$$

with precongruence \sqsubseteq'_o and congruence \equiv'_o .

Since \sqsubseteq (the order in the Plotkin powerdomain) is the Egli-Milner order, we can expand this definition: $P \leq' Q$ if and only if whenever $\langle P, M \rangle \rightarrow^* N$ then $\langle Q, M \rangle \rightarrow^* N$ and whenever $\langle Q, M \rangle \uparrow$ then $\langle P, M \rangle \uparrow$. Hence it is certainly the case that $P \leq' Q \Rightarrow P \leq Q$ and $P \sqsubseteq'_o Q \Rightarrow P \sqsubseteq_o Q$.

Now define a function $\text{flatten} : \rho \rightarrow (\mathcal{M}(\sigma) \rightarrow \mathcal{P}(\mathcal{M}(\sigma)))$ by $\text{flatten } f =$

```

 $\lambda s.$  over  $fs$  extend  $\{x\}$ .
  case  $x$  of
    inl( $(t, g)$ )  $\Rightarrow$  ( $\text{flatteng}$ ) $t$ ;
    inr( $t$ )  $\Rightarrow$   $\{t\}$ 
  end.
```

Proposition 1 For any program P , $\text{flatten}[[P]] = \mathcal{B}'(P)$.

Proof: Standard for resumption semantics.¹¹ □

Proposition 2 For any programs P and Q , $[[P]] \sqsubseteq [[Q]] \Rightarrow P \sqsubseteq'_o Q$.

Proof: The function flatten is continuous and in particular preserves order. The definition of $[[\cdot]]$ yields, for any context \mathbf{C} , a continuous function $[[\mathbf{C}[\cdot]]]$ on ρ . Hence we can carry out the following calculation.

$$\begin{aligned}
& \llbracket P \rrbracket \sqsubseteq \llbracket Q \rrbracket \\
\iff & \\
& \forall \mathbf{C}. \llbracket \mathbf{C}[P] \rrbracket \sqsubseteq \llbracket \mathbf{C}[Q] \rrbracket \\
\Rightarrow & \\
& \forall \mathbf{C}. \text{flatten}[\llbracket \mathbf{C}[P] \rrbracket] \sqsubseteq \text{flatten}[\llbracket \mathbf{C}[Q] \rrbracket] \\
\iff & \\
& \forall \mathbf{C}. \mathcal{B}'(\mathbf{C}[P]) \sqsubseteq \mathcal{B}'(\mathbf{C}[Q]) \\
\iff & \\
& P \sqsubseteq'_o Q
\end{aligned}$$

□

We have shown that the resumption semantics is sound with respect to the behavioural precongruence. However, the calculation in the proof cannot be reversed, and full abstraction fails. To see that the implication from $\forall \mathbf{C}. (\llbracket \mathbf{C}[P] \rrbracket \sqsubseteq \llbracket \mathbf{C}[Q] \rrbracket)$ to $\forall \mathbf{C}. (\text{flatten}[\llbracket \mathbf{C}[P] \rrbracket] \sqsubseteq \text{flatten}[\llbracket \mathbf{C}[Q] \rrbracket])$ cannot be reversed, consider the following programs.

$$\begin{array}{l}
P \quad x \rightarrow \{2\} \Leftarrow (x = 1) \\
Q \quad x, y \rightarrow \{2, 2\} \Leftarrow (x, y = 1, 1).
\end{array}$$

These programs might be written less formally as $1 \rightarrow 2$ and $1, 1 \rightarrow 2, 2$ respectively. Now consider the programs P and $P + Q$. With an initial multiset of integers, both of these programs will terminate after replacing every 1 by 2; the only difference is that $P + Q$ may terminate after fewer transitions, because the presence of Q allows two replacements to take place simultaneously.

It is fairly easy to convince oneself that P and $P + Q$ have the same behaviour in all contexts, and hence that $P \equiv_o P + Q$. This can be proved formally by means of the *transition trace semantics*,¹⁵ which equates P and $P + Q$ and is sound with respect to behavioural congruence. It is also clear that for any context \mathbf{C} and initial multiset M , $\langle P, M \rangle \uparrow \iff \langle P + Q, M \rangle \uparrow$, and so $P \equiv'_o P + Q$. However, their resumption semantics are different. If the initial multiset is $M = \{1, 1\}$, then because $P + Q$ can terminate after a single transition, $\llbracket P + Q \rrbracket M$ contains the terminal state $\{2, 2\}$. P , however, must make two transitions before terminating, and so $\llbracket P \rrbracket \neq \llbracket P + Q \rrbracket$. Hence we have $(\text{flatten}[\llbracket P \rrbracket])M = (\text{flatten}[\llbracket P + Q \rrbracket])M$ but $\llbracket P \rrbracket \neq \llbracket P + Q \rrbracket$, and the semantics is not fully abstract.

5 Reconstructing the Transition Assertions

In the framework of domain theory in logical form, each type σ has an associated logical theory $\mathcal{L}(\sigma)$. The formulae of $\mathcal{L}(\sigma)$ come from the language $L(\sigma)$. The $L(\sigma)$ are defined inductively by the following rules.

$$\begin{array}{c}
\frac{\phi_i \in L(\sigma)}{\bigvee_{i \in I} \phi_i, \bigwedge_{i \in I} \phi_i \in L(\sigma)} \qquad \frac{\phi \in L(\sigma) \quad \psi \in L(\tau)}{\phi \times \psi \in L(\sigma \times \tau)} \\
\\
\frac{\phi \in L(\sigma)}{\text{inl}(\phi) \in L(\sigma + \tau)} \qquad \frac{\psi \in L(\tau)}{\text{inr}(\psi) \in L(\sigma + \tau)} \\
\\
\frac{\phi \in L(\sigma)}{\Box \phi, \Diamond \phi \in L(\mathcal{P}(\sigma))} \qquad \frac{\phi \in L(\sigma) \quad \psi \in L(\tau)}{\phi \rightarrow \psi \in L(\sigma \rightarrow \tau)} \\
\\
\frac{\phi \in L(\sigma[(\text{rect } t.\sigma)/t])}{\phi \in L(\text{rect } t.\sigma)}
\end{array}$$

I is a finite indexing set. We write t and f for $\bigwedge_{i \in \emptyset} \phi_i$ and $\bigvee_{i \in \emptyset} \phi_i$ respectively.

The theory $\mathcal{L}(\sigma)$ has a relation \leq which corresponds semantically to implication, and a collection of axioms which give it the structure of a distributive

$$\begin{aligned}
\llbracket \phi \wedge \psi \rrbracket_\sigma &= \llbracket \phi \rrbracket_\sigma \cap \llbracket \psi \rrbracket_\sigma \\
\llbracket t \rrbracket_\sigma &= D(\sigma) \\
\llbracket \phi \vee \psi \rrbracket_\sigma &= \llbracket \phi \rrbracket_\sigma \cup \llbracket \psi \rrbracket_\sigma \\
\llbracket f \rrbracket_\sigma &= \emptyset \\
\llbracket \phi \times \psi \rrbracket_{\sigma \times \tau} &= \{(u, v) \mid u \in \llbracket \phi \rrbracket_\sigma, v \in \llbracket \psi \rrbracket_\tau\} \\
\llbracket \text{inl}(\phi) \rrbracket_{\sigma + \tau} &= \{\text{inl}(u) \mid u \in \llbracket \phi \rrbracket_\sigma\} \\
\llbracket \text{inr}(\psi) \rrbracket_{\sigma + \tau} &= \{\text{inr}(v) \mid v \in \llbracket \psi \rrbracket_\tau\} \\
\llbracket \phi \rightarrow \psi \rrbracket_{\sigma \rightarrow \tau} &= \{f \in D(\sigma \rightarrow \tau) \mid f(\llbracket \phi \rrbracket_\sigma) \subseteq \llbracket \psi \rrbracket_\tau\} \\
\llbracket \diamond \phi \rrbracket_{\mathcal{P}(\sigma)} &= \{S \in D(\mathcal{P}(\sigma)) \mid S \subseteq \llbracket \phi \rrbracket_\sigma\} \\
\llbracket \square \phi \rrbracket_{\mathcal{P}(\sigma)} &= \{S \in D(\mathcal{P}(\sigma)) \mid S \cap \llbracket \phi \rrbracket_\sigma \neq \emptyset\} \\
\llbracket \phi \rrbracket_{\text{rect.}\sigma} &= \{\beta_\sigma(u) \mid u \in \llbracket \phi \rrbracket_{\sigma[\text{rect.}\sigma/t]}\}
\end{aligned}$$

where $\beta_\sigma : D(\sigma[\text{rect.}\sigma/t]) \cong D(\text{rect.}\sigma)$ is the isomorphism arising from the solution of the domain equation $t = \sigma(t)$.

Figure 2: Semantics of formulae in $L(\sigma)$.

lattice and also describe the interaction between the lattice structure and the type constructors.

A formula in $L(\sigma)$ corresponds to a compact open set of elements of $D(\sigma)$. For each type σ , we define an interpretation function $\llbracket \cdot \rrbracket_\sigma : L(\sigma) \rightarrow K\Omega(D(\sigma))$. The definition appears in Figure 2, and follows that in Abramsky's work.³ Note that we take $D(\sigma + \tau) = \{\text{inl}(u) \mid u \in D(\sigma)\} \cup \{\text{inr}(v) \mid v \in D(\tau)\} \cup \{\perp\}$.

We will be interested in formulae in $L(\mathcal{P}(\alpha))$ of the following form:

$$\square \bigvee_{i \in I} \phi_i \wedge \bigwedge_{i \in I} \diamond \phi_i$$

for some indexing set I . Such a formula describes the following property of a set: any element must satisfy one of the ϕ_i , and for each ϕ_i there is an element satisfying it. Thus if $|I| = n$ then the set must contain n elements, each satisfying one of the ϕ_i .

A formula in $L(\rho)$ has the form $P \rightarrow \phi$ where $P \in L(\mathcal{M}(\sigma))$ and

$$\phi \in L(\mathcal{P}(\mathcal{M}(\sigma) \times \rho + \mathcal{M}(\sigma))).$$

Such a formula specifies that in a multiset satisfying P , the set of possible

actions satisfies ϕ . Now consider

$$\phi = \Box \bigvee_{i \in I} \phi_i \wedge \bigwedge_{i \in I} \Diamond \phi_i.$$

First suppose that $|I| > 1$. Then the formula $P \rightarrow \phi$ describes a set of several possibilities for a program executed in a multiset satisfying P . A Gamma program terminates only when there are no other possibilities, so every ϕ_i describes a pair consisting of a subsequent state and a resumption. Hence $\phi_i = \text{inl}(P_i \times \psi_i)$ with $P_i \in L(\mathcal{M}(\sigma))$ and $\psi_i \in L(\rho)$. Thus we have a formula

$$P \rightarrow \Box \bigvee_{i \in I} \text{inl}(P_i \times \psi_i) \wedge \bigwedge_{i \in I} \Diamond \text{inl}(P_i \times \psi_i)$$

which describes the possible non-terminal transitions of a program. This formula corresponds to the branching assertion $P \sum_{i \in I} P_i \psi_i$ in the transition assertion logic.

Now suppose that $|I| = 1$, so that the formula ϕ describes the unique transition of a program. Then

$$\phi = P \rightarrow \Box \phi_1 \wedge \Diamond \phi_1$$

which, because $\Box \psi \leq \Diamond \psi$ for any ψ , reduces to

$$\phi = P \rightarrow \Box \phi_1.$$

If the transition is non-terminal, we have $\phi = P \rightarrow \Box \text{inl}(P_1 \times \psi_1)$ as a special case of a branching assertion. If the transition is terminal, we can take $P_1 = P$ because the terminating step does not change the multiset. We then find that the formula $P \rightarrow \Box \text{inr}(P)$ corresponds to the terminal assertion $P \bullet$ of the transition assertion logic.

We have now recovered the terminal and branching assertions of the transition assertion logic. Conjunctions are automatically available. Recursive assertions are not supported by the domain theory in logical form framework, as their semantic interpretations may not be compact open sets, so we will work with finite unfoldings instead. Our modified language of transition assertions is defined by

$$\begin{array}{l} \phi ::= P \bullet \\ \quad | P \sum_{i \in I} P_i \phi_i \\ \quad | \phi \wedge \phi. \end{array}$$

6 The Proof System

Domain theory in logical form provides a typed metalanguage, which was used in Section 4 to define the resumption semantics of Gamma. There is also a proof system for statements of the form $M, \Gamma \vdash \phi$ in which M is a metalanguage term with some type σ , $\phi \in L(\sigma)$ is a logical formula, and $\Gamma = x_1 : \sigma_1, \dots, x_n : \sigma_n$ is a context of typed variables. The contexts Γ are necessary in proof rules such as

$$\frac{M, \Gamma[x \mapsto \phi] \vdash \psi}{\lambda x. M, \Gamma \vdash \phi \rightarrow \psi}$$

The transition assertion logic deals with statements of the form $C \text{ sat } \phi$ where C is a Gamma program and ϕ is a transition assertion. For each such statement, there is a corresponding statement $\llbracket C \rrbracket \vdash \phi'$ where $\llbracket C \rrbracket$ is the metalanguage term defining the semantics of C and $\phi' \in L(\rho)$ is the formula corresponding to the transition assertion ϕ . We now define the meaning of $C \text{ sat } \phi$ to be $\llbracket C \rrbracket \vdash \phi'$.

As in the original transition assertion logic we define parallel and sequential composition on transition assertions, such that the Gamma-level proof rules

$$\frac{C_1 \text{ sat } \phi \quad C_2 \text{ sat } \psi}{C_2 \circ C_1 \text{ sat } \psi \circ \phi} \quad \frac{C_1 \text{ sat } \phi \quad C_2 \text{ sat } \psi}{C_1 + C_2 \text{ sat } \phi \parallel \psi}$$

are justified. Now, however, the fact that these proof rules can be derived from the underlying system of axioms means that soundness of the logic follows from the general theory.

The assertions $\psi \circ \phi$ and $\phi \parallel \psi$ are defined inductively over the structure of ϕ and ψ . The definitions are slightly simpler than in the original formulation of the logic, because we are no longer using recursive assertions.

$$\begin{aligned} Q \bullet \circ P \bullet &= (P \wedge Q) \bullet \\ Q \sum_{j \in J} Q_j \psi_j \circ P \bullet &= (P \wedge Q) \sum_{j \in J} Q_j \psi_j \\ \psi \circ P \sum_{i \in I} P_i \phi_i &= P \sum_{i \in I} P_i (\psi \circ \phi_i) \\ (\psi_1 \wedge \psi_2) \circ \phi &= (\psi_1 \circ \phi) \wedge (\psi_2 \circ \phi) \\ \psi \circ (\phi_1 \wedge \phi_2) &= (\psi \circ \phi_1) \wedge (\psi \circ \phi_2) \\ \\ P \bullet \parallel Q \bullet &= (P \wedge Q) \bullet \\ P \bullet \parallel Q \sum_{j \in J} Q_j \psi_j &= (P \wedge Q) \sum_{j \in J} Q_j (P \bullet \parallel \psi_j) \\ P \sum_{i \in I} P_i \phi_i \parallel Q \bullet &= (P \wedge Q) \sum_{i \in I} P_i (\phi_i \parallel Q \bullet) \\ P \sum_{i \in I} P_i \phi_i \parallel Q \sum_{j \in J} Q_j \psi_j &= (P \wedge Q) (\sum_{i \in I} P_i (\phi_i \parallel Q \sum_{j \in J} Q_j \psi_j) \\ &\quad + \sum_{j \in J} Q_j (P \sum_{i \in I} P_i \phi_i \parallel \psi_j)) \\ \phi \parallel (\psi_1 \wedge \psi_2) &= (\phi \parallel \psi_1) \wedge (\phi \parallel \psi_2) \\ (\phi_1 \wedge \phi_2) \parallel \psi &= (\phi_1 \parallel \psi) \wedge (\phi_2 \parallel \psi) \end{aligned}$$

The Sequential Rule

We prove by induction on the structure of the assertions being combined that the proof rule can always be justified by a derivation in the domain logic. It turns out that for the purposes of the induction we need to justify a slightly more general proof rule; this is because when the inductive hypothesis (validity of the proof rule for smaller formulae) is applied, the statement being proved may include a context of assumptions about variables which will later disappear.

Proposition 3 If C_1 and C_2 are metalanguage terms corresponding to Gamma programs, and $\phi, \psi \in L(\rho)$ correspond to transition assertions, then the following proof rule can be derived:

$$\frac{C_1, \Gamma \vdash \phi \quad C_2, \Gamma \vdash \psi}{C_2 \circ C_1, \Gamma \vdash \psi \circ \phi.}$$

Proof: For each case in the definition of $\psi \circ \phi$ there is a derivation in the proof system of the domain logic. As an example, consider the case in which $\phi = P \sum_{i \in I} P_i \phi_i$. In terms of the domain logic,

$$\phi = P \rightarrow \square \bigvee_{i \in I} \text{inl}(P_i \times \phi_i) \wedge \bigwedge_{i \in I} \diamond \text{inl}(P_i \times \phi_i)$$

and

$$\psi \circ \phi = P \rightarrow \square \bigvee_{i \in I} \text{inl}(P_i \times (\psi \circ \phi_i)) \wedge \bigwedge_{i \in I} \diamond \text{inl}(P_i \times (\psi \circ \phi_i)).$$

Writing θ for $\square \bigvee_{i \in I} \text{inl}(P_i \times (\psi \circ \phi_i)) \wedge \bigwedge_{i \in I} \diamond \text{inl}(P_i \times (\psi \circ \phi_i))$, the derivation is

$$\frac{\begin{array}{cc} (1) & (2) \\ \hline \text{over } C_1 s \text{ extend } \{x\}. A, s \mapsto P \vdash \theta \end{array}}{\lambda s. \text{ over } C_1 s \text{ extend } \{x\}. A \vdash P \rightarrow \theta}$$

where (1) is the derivation in Figure 3, (2) is the derivation in Figure 4, and we are using the abbreviation

$$A = \text{ case } x \text{ of inl}((t, h)) \Rightarrow \{\text{inl}((t, h; C_2))\}; \text{ inr}(t) \Rightarrow C_2 s \text{ end.}$$

$$\begin{array}{c}
\begin{array}{c}
C_1 \vdash P \rightarrow \square \bigvee_{i \in I} \text{in}(P_i \times \phi_i) \wedge \bigwedge_{i \in I} \Diamond \text{in}(P_i \times \phi_i) \\
\vdots \\
C_1 s, s \vdash P \vdash \Diamond \text{in}(P_i \times \phi_i)
\end{array}
\quad
\begin{array}{c}
\text{axiom} \\
\hline
x, x \mapsto \text{in}(P_i \times \phi_i) \vdash \text{in}(P_i \times \phi_i) \\
\hline
x, x \mapsto \text{in}(P_i \times \phi_i), s \mapsto P \vdash \text{in}(P_i \times \phi_i) \\
\hline
A, s \mapsto P, x \mapsto \bigvee_{i \in I} \text{in}(P_i \times \phi_i) \vdash \Diamond \text{in}(P_i \times (\psi \circ \phi_i)) \\
\hline
\text{over } C_1 s \text{ extend } \{x\}. A, s \mapsto P \vdash \Diamond \text{in}(P_i \times (\psi \circ \phi_i)) \\
\hline
\text{over } C_1 s \text{ extend } \{x\}. A, s \mapsto P \vdash \bigwedge_{i \in I} \Diamond \text{in}(P_i \times (\psi \circ \phi_i))
\end{array}
\quad
\begin{array}{c}
\text{axiom} \\
\hline
t, t \mapsto P \vdash P \\
\hline
t, t \mapsto P, h \mapsto \phi_i \vdash P \\
\hline
(t, C_2 \circ h), t \mapsto P_i, h \mapsto \phi_i \vdash P_i \times (\psi \circ \phi_i) \\
\hline
\text{in}((t, C_2 \circ h)), t \mapsto P_i, h \mapsto \phi_i \vdash \text{in}(P_i \times (\psi \circ \phi_i)) \\
\hline
\{\text{in}((t, C_2 \circ h))\}, t \mapsto P_i, h \mapsto \phi_i \vdash \Diamond \text{in}(P_i \times (\psi \circ \phi_i))
\end{array}
\quad
\begin{array}{c}
\text{given} \\
\hline
C_2 \vdash \psi \\
\hline
\text{axiom} \\
\hline
h, h \mapsto \phi_i \vdash \phi_i \quad C_2, h \mapsto \phi_i \vdash \psi \\
\hline
C_2 \circ h, h \mapsto \phi_i \vdash \psi \circ \phi_i \\
\hline
C_2 \circ h, t \mapsto P_i, h \mapsto \phi_i \vdash \psi \circ \phi_i \\
\hline
\end{array}
\end{array}$$

Figure 4: Derivation of Right Conjunction of θ .

The Parallel Rule

Proposition 4 If C_1 and C_2 are metalanguage terms corresponding to Gamma programs, and $\phi, \psi \in L(\rho)$ correspond to transition assertions, then the following proof rule can be derived.

$$\frac{C_1, \Gamma \vdash \phi \quad C_2, \Gamma \vdash \psi}{C_1 + C_2, \Gamma \vdash \phi \parallel \psi}$$

Basic Reactions

In the original formulation of the transition assertion logic for Gamma, the proof rule for a basic reaction establishes a recursive assertion. Now we use two proof rules which between them allow any finite unfolding of the original recursive assertion to be proved.

$$\frac{(A \Leftarrow R), \Gamma \vdash \neg \exists \bar{x}. R(\bar{x}) \bullet \quad (A \Leftarrow R), \Gamma \vdash \phi \quad \{P\}A(\bar{x})/\bar{x}\{P\}}{(A \Leftarrow R), \Gamma \vdash (P \wedge \exists \bar{y}. R(\bar{y})) \sum (P \wedge \exists \bar{y}. \bar{y} = A(\bar{x}))\phi}$$

In the second rule, the branch is over all possible ways of instantiating \bar{y} to a tuple satisfying R .

To justify these proof rules, we need to add rules for the metalanguage terms corresponding to multiset manipulations.

Proposition 5 If the following rules are added to the domain logic proof system, then the above rules for basic reactions can be derived.

$$\frac{\neg(\exists \bar{x} \in M. R(\bar{x}))}{\text{test } M \text{ for } R, \Gamma \vdash \text{false}} \quad \frac{\exists \bar{x} \in M. R(\bar{x})}{\text{test } M \text{ for } R, \Gamma \vdash \text{true}}$$

$$\text{substitute } A(\bar{x}) \text{ for } \bar{x} \text{ in } M, \Gamma \vdash \exists \bar{y}. \bar{y} = A(\bar{x})$$

$$\frac{\{\phi\}A(\bar{x})/\bar{x}\{\psi\} \quad M, \Gamma \vdash \phi}{\text{substitute } A(\bar{x}) \text{ for } \bar{x} \text{ in } M, \Gamma \vdash \psi} \quad \frac{s, \Gamma \vdash \exists \bar{x}. R(\bar{x})}{\text{extract } R \text{ from } s, \Gamma \vdash \square R}$$

7 Example

Having recovered a modified form of the transition assertion logic, we will now demonstrate that it can still be used for the same purposes as the original.

The example we consider is the Gamma program for calculating Fibonacci numbers, as described in Section 1.

Recall the definitions

$$\begin{aligned} C_1 &= x \rightarrow \{x-1, x-2\} \Leftarrow (x > 1) \\ C_2 &= x \rightarrow \{1\} \Leftarrow (x = 0) \\ C_3 &= x, y \rightarrow \{x+y\} \Leftarrow \text{true} \end{aligned}$$

such that the program $C = C_3 \circ (C_1 + C_2)$ 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 this fact. The structure of the proof is exactly the same as the original.⁷ The only difference is that because recursive assertions are no longer available, we work explicitly with finite unfoldings.

Consider the following assertions about multisets M .

1. $P_1 = (\sum_{m \in M} f_m = f_{n_0})$
2. $P_2 = (\sum_{m \in M} m = f_{n_0})$.

We assert that P_1 is an invariant for C_1 and C_2 , and P_2 is an invariant for C_3 . Formally proving such assertions would require a formal system for reasoning about multisets; this is one of the issues discussed in the next Section. Proving them informally is trivial.

Define a sequence of assertions ϕ_n by

$$\begin{aligned} \phi_0 &= (P_1 \wedge \forall x. x \leq 1) \bullet \\ \phi_{n+1} &= (P_1 \wedge \forall x. x \leq 1) \bullet \wedge (P_1 \wedge \exists x. x > 1) \sum_{i \in 1} P_1 \phi_n. \end{aligned}$$

In these and subsequent definitions, all quantification is over the elements of the current multiset. For each n we can prove $C_1 \text{ sat } \phi_n$. Similarly, defining the assertions ψ_n by

$$\begin{aligned} \psi_0 &= (P_1 \wedge \forall x. x \neq 0) \bullet \\ \psi_{n+1} &= (P_1 \wedge \forall x. x \neq 0) \bullet \wedge (P_1 \wedge \exists x. x = 1) \sum_{i \in 1} P_1 \psi_n \end{aligned}$$

we can prove for each n , $C_2 \vdash \psi_n$. The same goes for C_3 with the assertions θ_n defined by

$$\begin{aligned} \theta_0 &= (P_2 \wedge \neg \exists \langle x, y \rangle) \bullet \\ \theta_{n+1} &= (P_2 \wedge \neg \exists \langle x, y \rangle) \bullet \wedge (P_2 \wedge \exists \langle x, y \rangle) \sum_{i \in 1} P_2 \theta_n. \end{aligned}$$

Using the proof rules, we can prove that for any m, n and r , $C_3 \circ (C_1 + C_2) \vdash \theta_r \circ (\phi_m \parallel \psi_n)$. We now need to do some calculation in order to extract more information from this fact.

The assertion $\phi_m \parallel \psi_n$ describes the possible terminating executions of $C_1 + C_2$. Each possibility ends with the assertion

$$(P_1 \wedge (\forall x. x \leq 1) \wedge (\forall x. x \neq 0)) \bullet.$$

Hence the program $C_1 + C_2$ can terminate with a multiset M satisfying

$$\left(\sum_{m \in M} f_m = f_{n_0} \right) \wedge (\forall x. x = 1).$$

Since $f_1 = 1$, this implies P_2 , the precondition for C_3 . Now we use the fact that $C_3 \vdash \theta_r$ for any r . The assertion θ_r always contains the conjunct $(P_2 \wedge \neg \exists \langle x, y \rangle) \bullet$. Hence C_3 can terminate with a multiset M satisfying $(\sum_{m \in M} m = f_{n_0}) \wedge \neg \exists \langle x, y \rangle$. Thus the final multiset must be the singleton $\{\!\{f_{n_0}\}\!\}$.

8 Soundness and Completeness

Having described a language of assertions and a proof system, we will now discuss the questions of soundness and completeness.

Let P be a Gamma program, let P' be the corresponding metalanguage term, and let $\llbracket P' \rrbracket \in D(\rho)$ be the semantics of that metalanguage term. Also let $\phi \in L(\rho)$ be a formula corresponding to a transition assertion, and as usual let $\llbracket \phi \rrbracket_\rho$ be its semantics. By the soundness of the proof system for the metalanguage of domain theory in logical form, we have

Proposition 6 $P \vdash \phi \Rightarrow \llbracket P' \rrbracket \in \llbracket \phi \rrbracket_\rho$.

The definition of $\llbracket \phi \rrbracket_\rho$ for a transition assertion ϕ means that if $\llbracket P' \rrbracket \in \llbracket \phi \rrbracket_\rho$ then the behaviour of P is as described by the structure of ϕ . Hence, writing $P \mathbf{sat} \phi$ for operational satisfaction in the sense of Section 2, we have

Proposition 7 [Soundness] $\forall P, \phi. (P \vdash \phi \Rightarrow P \mathbf{sat} \phi)$.

Because $(\forall \phi. x \in \llbracket \phi \rrbracket_\rho \Rightarrow y \in \llbracket \phi \rrbracket_\rho) \Rightarrow (x \sqsubseteq y)$, we also have

Proposition 8 $\forall \phi. (P \vdash \phi \Rightarrow Q \vdash \phi) \Rightarrow (\llbracket P \rrbracket \sqsubseteq \llbracket Q \rrbracket)$.

Corollary 9 $\forall \phi. (P \vdash \phi \Rightarrow Q \vdash \phi) \Rightarrow (\llbracket P \rrbracket \sqsubseteq_o \llbracket Q \rrbracket)$.

We will now consider completeness, which we have not obtained. If we could present the entire proof system for the transition assertion logic in terms of the domain logic, then we could deduce the converse of Proposition 8:

$$\forall \phi. (P \vdash \phi \Rightarrow Q \vdash \phi) \Leftarrow (\llbracket P \rrbracket \sqsubseteq \llbracket Q \rrbracket)$$

in other words, the logic would exactly characterise denotational approximation.

Unfortunately, two things go wrong. First, to get the correspondence between the proof system and the resumption semantics we need the full power of Stone Duality from domain theory in logical form. To establish the duality between the semantics and the logic we would need a sound and complete system for reasoning about multisets. Not only have we not exhibited such a system, but there are theoretical reasons why this may not be possible at all within the domain logic framework. This is because of the necessity of using the negation of a reaction condition in the proof rule for a basic reaction.

Second, the failure of full abstraction for the resumption semantics means that we only have $P \sqsubseteq Q \Rightarrow P \sqsubseteq_o Q$ and not vice versa. Even if the logic characterised denotational approximation, we would still not obtain the desired characterisation of operational approximation; for example, it would not be possible to use the logic to prove $\neg(P \sqsubseteq_o Q)$ by finding an assertion satisfied by P but not by Q .

9 Conclusions

Starting with a denotational semantics of Gamma, in the resumption style, we have applied Abramsky’s framework of domain theory in logical form to obtain a logic of Gamma programs—the transition assertion logic. This logic was originally formulated for Gamma by Errington *et al.*, and was based on previous work by Brookes on logics for shared-variable parallel languages. Domain theory in logical form has already been applied to such languages by Zhang; he recovers the formulae of the transition assertion logic from a resumption semantics. However, we have gone further by demonstrating that the proof rules for transition assertions can be justified as derivations in the logic corresponding to the resumption semantics.

Because domain theory in logical form does not support recursively-defined formulae, we actually get a slightly modified form of the transition assertion logic in which recursive formulae are replaced with sequences of finite unfoldings. As an example of the use of this modified logic, we have given a correctness proof for a simple Gamma program.

As mentioned in Section 8, we do not get the full benefit of domain theory in logical form. This is for two reasons: the failure of full abstraction for the resumption semantics, and the lack of a formalised system for reasoning about properties of multisets. To address the first problem, we need a fully abstract semantics of Gamma, in a form suitable for the application of Abramsky’s theory. One possibility is to adapt the work of Horita *et al.*,^{1,2} which uses

metric spaces as a basis for fully abstract models of shared-variable parallel languages. The idea behind their semantics is the transition trace semantics of Brookes.⁶ Since a similar semantics has been studied for Gamma,¹⁵ this avenue seems quite promising. Domain theory in logical form is intended to be general enough to deal with metric space semantics as well as domain-theoretic semantics, although we do not know whether this generalisation has been developed.

Probably the best way to obtain a suitable formal system for reasoning about multisets is to use a general construction for multisets over arbitrary domains, and formulate the corresponding axioms for the domain logic. However, at the moment we do not know of a multiset domain construction. Also, the use of negation in the proof rule for a basic reaction may be difficult to reconcile with the intuitionistic style of the domain logic. It is not clear what will be the best approach to overcoming this difficulty.

We have two main applications in mind for the programme of work begun in this paper. The first is simply that if reasoning about Gamma programs is formalised, we can begin to investigate automating the verification process. Secondly, we would like to do something similar to Jensen's work on strictness logic.¹³ He shows that if abstract domains for strictness analysis are used, the domain logic yields a formal system for reasoning about strictness properties. It would be interesting to adapt his work from its original setting of functional languages to Gamma.

Acknowledgement

This research was supported by the Esprit Basic Research Action 9102 (Coordination).

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 and D. Le Métayer. The GAMMA model and its discipline of programming. *Science of Computer Programming*, 15:55–77, 1990.

5. S. D. Brookes. An axiomatic treatment of a parallel programming language. In R. Parikh, editor, *Logics of Programs*, volume 193 of *LNCS*. Springer-Verlag, 1985.
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. 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.
8. C. L. Hankin, D. Le Métayer, and D. Sands. A calculus of Gamma programs. Technical Report 672, INRIA-RENNES, October 1992.
9. C. L. Hankin, D. Le Métayer, and D. Sands. The coordination language Gamma: Semantics and transformation. Submitted for publication, 1995.
10. M. Hennessy and R. Milner. Algebraic laws for non-determinism and concurrency. *JACM*, 32:137–161, 85.
11. 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.
12. E. Horita, J. W. de Bakker, and J. J. M. M. Rutten. Fully abstract denotational models for nonuniform concurrent languages. *Information and Computation*, 115:125–178, 1994.
13. T. P. Jensen. Strictness analysis in logical form. In *FPCA91*, 1991.
14. G. Plotkin. Post-graduate lecture notes in advanced domain theory (incorporating the “Pisa Notes”). Dept. of Computer Science, Univ. of Edinburgh, 1981.
15. 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.
16. G.-Q. Zhang. *Logic of Domains*. Progress in Theoretical Computer Science. Birkhauser, 1990.