2 The Computational Model

The aim of this thesis is to develop a methodology which supports the separate design of the computation and coordination aspects of programs. In order to realize this approach, we need a programming model that supports this separation between computation and coordination. Existing programming models usually stress only one of these aspects. For instance, functional and logical programming languages emphasize their declarative nature and the advantages it has for proving correctness, but deny the programmer effective means for determining the program's behaviour. With imperative languages the programmer has complete control over the operational behaviour of his program. However, because the control-flow is an integral part of imperative programs, it is difficult to focus on correctness while abstracting from operational details.

In this chapter we will present the Gamma programming model which has shown to be well suited for specifying the computation component of a program without imposing premature constraints on the coordination component. We present a concise semantics for Gamma programs and a formal logic for reasoning about their correctness.

2.1 The Gamma Programming Model

We start with a brief introduction to Gamma. For more details the reader is referred to [13] which includes a series of example programs.

The uniform data structure in Gamma is the multiset. Multisets can be formed over arbitrary domains of values, including integers, reals, booleans and tuples. The simplest Gamma program is a conditional multiset rewrite-rule, written as $\overline{x} \mapsto m \Leftarrow b$. Here $\overline{x}$ denotes a sequence of variables $x_1, \ldots, x_n$, m is a multiset expression, and b is a boolean expression. The free variables that occur in m and b are taken from $x_1, \ldots, x_n$.

Application of the rule $\overline{x} \mapsto m \Leftarrow b$ to a multiset proceeds by replacing elements in the multiset satisfying the condition b by the elements that result from evaluating the multiset expression m. This step is repeated until no more elements are present that satisfy b. The resulting multiset denotes the outcome of the program.

Example 2.1.1 *We introduce a Gamma program for sorting a sequence of numbers into ascending order. The input sequence is represented by a multiset consisting of value-index pairs. The Gamma program consists of a single multiset rewrite rule which is defined as follows*

$$swap \mathrel{\widehat{=}} (i,x),(j,y) \mapsto (i,y),(j,x) \Leftarrow x > y \;\wedge\; i < j \tag{2.1}$$

The program executes by exchanging ill-ordered values until there are no more pairs that satisfy this condition. At that point the resulting multiset represents a well-ordered sequence. Disjoint pairs can be compared and exchanged in parallel, but this need not necessarily be the case.

A possible execution for the sorting program is depicted in Figure 2.1.

Figure 2.1: Possible execution of the program swap *in a multiset* $M_0 = \{(1,D),(2,C),(3,B),(4,A)\}$

It is important to note that the Gamma program does not specify in which order pairs of values are compared and exchanged. Hence the program can be seen as a highly nondeterministic specification of a wide spectrum of sorting strategies. This opens up the opportunity for a separate specification of the operational aspects of the program. This gap will be filled by the coordination language that we will present in Chapter 3.

More complex Gamma programs can be built using two basic combinators. Individual rules can be composed into so-called *simple* programs [65] using the parallel combinator, denoted "+". The constituent rules in parallel composition are executed in any order (possibly in parallel) until none of the rules can be successfully applied.

Simple programs can in turn be composed using the sequential combinator , denoted " $\circ$ ". If P_1 and P_2 are simple programs, then $P_1 \circ P_2$ first executes P_2 until its rules can no longer be applied, after which P_1 is executed on the resulting multiset.

The abstract syntax of Gamma programs can be specified as follows. We use r, R and P to range over the syntactic categories of multiset rewrite-rules, simple programs, and programs respectively. We use $\mathbb{P}$ to denote the set of all Gamma programs.

Syntactic Categories

$$\begin{array}{lll} r & \in & \mathit{Rule} \\ R & \in & \mathit{Simple} \\ P & \in & \mathit{Program} \end{array}$$

Definition

$$\begin{array}{lllll} r & ::= & \overline{x} \mapsto m \Leftarrow b & & \\ R & ::= & r & \mid & R + R \\ P & ::= & R & \mid & P \circ P \end{array}$$

Figure 2.2: Abstract Syntax of Multiset Transformer Programs

The program terms derivable in this way are "products of sums"; i.e. are of the form $(r_1 + \cdots + r_i) \circ \cdots \circ (r_j + \cdots + r_n)$. The purpose of limiting the syntax of program terms to this form is to exclude the parallel composition of programs that contain sequential composition; e.g. $P_1 + (P_2 \circ P_3)$. There are two reasons for excluding these forms: firstly, the syntax thus obtained describes exactly the same set of programs that are definable by the original Gamma model presented in [12] and [13]. Secondly, the excluded terms present difficulties with the compositionality of semantics [34]. An additional justification for the focus on programs in product-of-sums form is a result of Sands [106] which entails that every Gamma program can be refined by a program that is in product-of-sums form.

We use the method of *structural operational semantics* [96] to define the meaning of Gamma programs. To this end we introduce configurations, denoted $\langle P, M\rangle$, where P is a Gamma program and M is a multiset. A configuration represents the state of a computation. A configuration can move to another configuration by performing an action. Such actions are modelled by a relation between configurations. To define the semantics of Gamma, we use a labelled multi-step transition relation.

A transition is written as $\langle P, M\rangle \overset{\sigma}{\rightsquigarrow} \langle P', M'\rangle$ where the label σ is a multiset substitution which formally describes the rewrite action that transforms M into M'. A terminal configuration is written $\langle P, M\rangle\surd$.

The semantics of Gamma is collected in Figure 2.3. The multi-step transition relation is defined in terms of a single-step transition relation. The latter is distinguished from the multi-step transition relation by decorating it with a subscript "1": $\overset{\sigma}{\rightsquigarrow}_1$. This single-step transition relation will be used in Chapter 3 to link the semantics of the coordination component to that of the computation component.

The various notations that we use in defining the semantics are best explained by considering the semantic rule for execution of an individual rewrite rule $r = \overline{x} \mapsto m \Leftarrow b$:

$$\text{if } \overline{v} \subseteq M : b[\overline{x} := \overline{v}] \text{ then } \langle r, M\rangle \overset{\sigma}{\rightsquigarrow}_1 \langle r, M[\sigma]\rangle \text{ where } \sigma = m[\overline{x} := \overline{v}]/\overline{v}$$

We write $b[\overline{x} := \overline{v}]$ to denote the boolean expression that results from replacing each free occurrence of x_i by v_i. We write $\sigma = M/N$ to denote a multiset substitution σ which replaces N by M. By $M[\sigma]$ we denote the multiset that results from applying the substitution σ to M. More formally, let $M' = m[\overline{x} := \overline{v}]$, then $M[M'/\overline{v}] = (M \ominus \overline{v}) \oplus M'$, where $\oplus$ and $\ominus$ denote multiset addition and subtraction respectively (their formal definition can be found in Appendix A.2). Note that for ease of notation we confuse the sequence $\overline{v}$ with the multiset consisting of the same elements as $\overline{v}$.

When multiple transitions transform disjoint parts of the multiset, then these transitions do not interfere with each other, hence they can also happen in parallel. This observation directly leads to the multi-step transition semantics of Gamma, in particular semantic rule (C4), as defined in Figure 2.3.

We present two variants of a formal definition of non-interference . The first notion is the most strict: it requires the elements that are retrieved from the multiset to be strictly disjoint. The second is more flexible: it allows elements to be read[1] concurrently by multiple multiset substitutions. This difference corresponds to *exclusive read/exclusive write*

[1]The removal and insertion of identical elements by a single multiset-substitution is interpreted as reading of those elements.

and *concurrent read/exclusive write* mechanisms found in the classification of architectures for parallel computers. By default, we use Definition 2.1.3.

Definition 2.1.2 *Given a multiset M and two multiset substitutions $\sigma_1 = M_1/N_1$ and $\sigma_2 = M_2/N_2$, we say that σ_1 and σ_2 are* independent *in M, denoted $M \models \sigma_1 \bowtie_E \sigma_2$, if $N_1 \oplus N_2 \subseteq M$.*

Definition 2.1.3 *Given a multiset M and two multiset substitutions $\sigma_1 = M_1/N_1$ and $\sigma_2 = M_2/N_2$.*

1. *We say that σ_1 is* independent from *σ_2 in M, denoted $M \models \sigma_1 \lhd \sigma_2$, if $N_1 \subseteq (M \ominus N_2) \cup M_2$.*
2. *We write $M \models \sigma_1 \bowtie \sigma_2$ if σ_1 and σ_2 are* mutually independent *in M; i.e. if $M \models \sigma_1 \lhd \sigma_2$ and $M \models \sigma_2 \lhd \sigma_1$*

The label assigned to a multi-step transition is a combination of the labels of the constituent transitions. The concurrence of multiple multiset substitutions can be formally described using the composition operator.

Definition 2.1.4 *Given two multiset substitutions $\sigma_1 = M_1/N_1$ and $\sigma_2 = M_2/N_2$, the* composition *of σ_1 and σ_2 is defined as $\sigma_1 \cdot \sigma_2 = (M_1 \oplus M_2)/(N_1 \oplus N_2)$.*

$$\text{(C0)} \qquad \langle \overline{x} \mapsto m \Leftarrow b, M \rangle \surd \qquad \text{if } \neg(\exists \overline{v} \subseteq M : b[\overline{x} := \overline{v}])$$

$$\text{(C1)} \qquad \langle \overline{x} \mapsto m \Leftarrow b, M \rangle \overset{\sigma}{\rightsquigarrow}_1 \langle \overline{x} \mapsto m \Leftarrow b, M[\sigma] \rangle \qquad \begin{array}{l} \text{if } \overline{v} \subseteq M \wedge b[\overline{x} := \overline{v}] \\ \text{where } \sigma = m[\overline{x} := \overline{v}]/\overline{v} \end{array}$$

$$\text{(C2)} \qquad \frac{\langle R, M \rangle \overset{\sigma}{\rightsquigarrow}_1 \langle R, M' \rangle}{\langle R, M \rangle \overset{\sigma}{\rightsquigarrow} \langle R, M' \rangle}$$

$$\text{(C3)} \qquad \frac{\langle R_1, M \rangle \overset{\sigma}{\rightsquigarrow}_1 \langle R_1, M' \rangle}{\begin{array}{c} \langle R_1 + R_2, M \rangle \overset{\sigma}{\rightsquigarrow}_1 \langle R_1 + R_2, M' \rangle \\ \langle R_2 + R_1, M \rangle \overset{\sigma}{\rightsquigarrow}_1 \langle R_2 + R_1, M' \rangle \end{array}}$$

$$\text{(C4)} \qquad \frac{\begin{array}{c} \langle R, M \rangle \overset{\sigma_1}{\rightsquigarrow}_1 \langle R, M_1 \rangle \\ \langle R, M \rangle \overset{\sigma_2}{\rightsquigarrow} \langle R, M_2 \rangle \end{array}}{\langle R, M \rangle \overset{\sigma_1 \cdot \sigma_2}{\rightsquigarrow} \langle R, M[\sigma_1 \cdot \sigma_2] \rangle} \qquad \text{if } M \models \sigma_1 \bowtie \sigma_2$$

$$\text{(C5)} \qquad \frac{\begin{array}{c} \langle R_1, M \rangle \surd \\ \langle R_2, M \rangle \surd \end{array}}{\langle R_1 + R_2, M \rangle \surd}$$

$$\text{(C6)} \qquad \frac{\begin{array}{c} \langle P_1, M \rangle \surd \\ \langle P_2, M \rangle \overset{\sigma}{\rightsquigarrow} \langle P_2', M' \rangle \end{array}}{\langle P_2 \circ P_1, M \rangle \overset{\sigma}{\rightsquigarrow} \langle P_2', M' \rangle}$$

$$\text{(C7)} \qquad \frac{\langle P_1, M \rangle \overset{\sigma}{\rightsquigarrow} \langle P_1', M' \rangle}{\langle P_2 \circ P_1, M \rangle \overset{\sigma}{\rightsquigarrow} \langle P_2 \circ P_1', M' \rangle}$$

$$\text{(C8)} \qquad \frac{\begin{array}{c} \langle P_1, M \rangle \surd \\ \langle P_2, M \rangle \surd \end{array}}{\langle P_1 \circ P_2, M \rangle \surd}$$

Figure 2.3: Semantics of Multiset Transformer Programs

The semantics of Gamma as defined in Figure 2.3 differs from the one presented in [65]. The latter uses a single-step transition relation which suggests an interleaved semantics.

The idea behind the coordination language that we will present in Chapter 3 is that it restricts the otherwise nondeterministic behaviour of Gamma programs, hence it cannot introduce new behaviour. Consequently, the semantics we choose for programs, limits the behaviours we can obtain using a coordination language. Because we want to distinguish between parallel and sequential execution at the coordination level, we need this distinction to be present in the semantics of Gamma.

In Section 9.2.3 of Chapter 9 we will describe a technical anomaly of single-step semantics that occurs in the context of refinement. The fact that multi-step semantics does not exhibit this anomaly is another reason for preferring it over single-step semantics.

The multi-step operational semantics of Figure 2.3 and the single-step operational semantics of [65] endow Gamma programs with different behaviour (in the sense of the possible (sequences of) transitions), but induce the same functionality (input-output relation) for programs.

To prove the functional equivalence between the multi-step and single-step semantics, we formalize the notion of input-output relation. To this end, we first define the reflexive transitive closure of the transition relation and a "may diverge" predicate.

Definition 2.1.5 *We define the reflexive transitive closure of the transition relation, denoted $\rightsquigarrow^*$, by*

$$\langle P, M\rangle \overset{\langle\,\rangle}{\rightsquigarrow}{}^* \langle P, M\rangle \qquad \frac{\langle P, M\rangle \overset{\sigma}{\rightsquigarrow} \langle P', M'\rangle}{\langle P, M\rangle \overset{\sigma}{\rightsquigarrow}{}^* \langle P', M'\rangle} \qquad \frac{\begin{array}{c}\langle P, M\rangle \overset{\overline{\sigma_1}}{\rightsquigarrow}{}^* \langle P', M'\rangle \\ \langle P', M'\rangle \overset{\overline{\sigma_2}}{\rightsquigarrow}{}^* \langle P'', M''\rangle\end{array}}{\langle P, M\rangle \overset{\overline{\sigma_1}\cdot\overline{\sigma_2}}{\rightsquigarrow}{}^* \langle P'', M''\rangle}$$

The reflexive transitive transition relation uses labels $\overline{\sigma}$ which denote sequences of individual labels. For convenience we identify the singleton sequence $\langle\, \sigma\, \rangle$ with its only element σ.

Definition 2.1.6 *A configuration $\langle P, M\rangle$ may diverge, denoted $\langle P, M\rangle\uparrow$, if and only if $\langle P, M\rangle = \langle P_0, M_0\rangle$ and for all $i \geq 0$ there exists a σ_i such that $\langle P_i, M_i\rangle \xrightarrow{\sigma_i} \langle P_{i+1}, M_{i+1}\rangle$.*

Definition 2.1.7 *The capability function for programs $\mathcal{C} : \mathbb{P} \times \mathbb{M} \to \mathcal{P}(\mathbb{M}) \cup \{\perp\}$ is* *defined as*

$$\mathcal{C}(P,M) = \ \{\bot \mid \langle P,M\rangle\uparrow\} \ \cup \ \{M' \mid \langle P,M\rangle \overset{\bar{\sigma}}{\rightsquigarrow}{}^{*} \langle P',M'\rangle\surd\}$$

Example 2.1.8 *Consider the sorting program from Example 2.1.1 and an initial sequence* $\langle\, 13,7,97\,\rangle$. *Then* $\mathcal{C}(\mathit{swap}, \{(1,13),(2,7),(3,97)\}) = \{\{(1,7),(2,13),(3,97)\}\}$.

We show the functional equivalence of the multi-step and single-step semantics for simple programs. The generalization to arbitrary Gamma programs is straightforward.

The multi-step semantics for simple Gamma program consists of the inference rules (C0), (C1), (C2), (C3), (C4) and (C5) from Figure 2.3. The single-step semantics consist of inference rules (C0), (C1), (C2), (C3) and (C5). We use $\mathcal{C}_1$ to denote the capability function for the single-step semantics.

First, we show that for every multi-step transition there exists a sequence of single-step transitions, denote $\rightsquigarrow_1{}^*$, that has the same effect on the multiset.

Many of the lemmas in this thesis, for example Lemma 2.1.9, are of the form "if some transition $\langle P,M\rangle \overset{\sigma}{\rightsquigarrow} \langle P',M'\rangle$, then some conclusion". The method of structural operational semantics [96] ensures that every transition is derived by a finite number of inferences using the semantic rules. This allows statements of the aforementioned type, to be proven by induction on the depth of this inference tree. This method is called *proof by transition induction* . This technique is used the proof of Lemma 2.1.9.

Lemma 2.1.9 *Let* P *be a simple program. If* $\langle P,M\rangle \overset{\sigma}{\rightsquigarrow} \langle P,M'\rangle$, *then there exists a sequence of single-step transitions*

$$\langle P,M_0\rangle \overset{\sigma_1}{\rightsquigarrow}_1 \langle P,M_1\rangle \ldots \overset{\sigma_i}{\rightsquigarrow}_1 \ldots \langle P,M_{n-1}\rangle \overset{\sigma_n}{\rightsquigarrow}_1 \langle P,M_n\rangle$$

such that $M_0 = M$ *and* $M_n = M'$ *and* $\sigma = \sigma_1 \cdot \ldots \cdot \sigma_n$.

Proof By transition induction: consider the possible ways in which the last inference of the transition $\langle P,M\rangle \overset{\sigma}{\rightsquigarrow} \langle P,M'\rangle$ may have been made.

- by (C2) from $\langle P,M\rangle \overset{\sigma}{\rightsquigarrow}_1 \langle P,M'\rangle$. Then the result holds directly.
- by (C4) from $\langle P,M\rangle \overset{\sigma_1}{\rightsquigarrow}_1 \langle P,M''\rangle$ and $\langle P,M\rangle \overset{\sigma_2}{\rightsquigarrow} \langle P,M'''\rangle$ where $M \models \sigma_1 \bowtie \sigma_2$. From Lemma A.2.6 follows that these transitions may be applied in arbitrary interleaved order; for instance

$$\langle P,M\rangle \overset{\sigma_1}{\rightsquigarrow}_1 \langle P,M''\rangle \quad \text{and} \quad \langle P,M''\rangle \overset{\sigma_2}{\rightsquigarrow} \langle P,M'\rangle$$

By the induction hypothesis we get for the latter transition that there exists a sequence of single-step transitions

$$\langle P, M_0\rangle \overset{\sigma'_1}{\rightsquigarrow}_1 \langle P, M_1\rangle \ldots \overset{\sigma'_i}{\rightsquigarrow}_1 \ldots \langle P, M_{n-1}\rangle \overset{\sigma'_n}{\rightsquigarrow}_1 \langle P, M_n\rangle$$

such that $M_0 = M''$ and $M_n = M'''$ and $\sigma_2 = \sigma'_1 \cdot \ldots \cdot \sigma'_n$. The result follows from concatenation of this sequence of single-step transitions to $\langle P, M\rangle \overset{\sigma_1}{\rightsquigarrow}_1 \langle P, M''\rangle$.

□

Theorem 2.1.10 *Let P be a simple program. Then, $\forall M : \mathcal{C}(P, M) = \mathcal{C}_1(P, M)$.*

Proof We prove that $\mathcal{C}(P, M) \subseteq \mathcal{C}_1(P, M)$ and $\mathcal{C}_1(P, M) \subseteq \mathcal{C}(P, M)$.

- $\mathcal{C}(P, M) \subseteq \mathcal{C}_1(P, M)$: By Lemma 2.1.9 follows that every multi-step transition can be mimicked by a sequence of single-step transitions. By induction on the length of the transition sequence follows that the single-step semantics can mimic any sequence of multi-step transitions (be it a finite or infinite sequence).

- $\mathcal{C}_1(P, M) \subseteq \mathcal{C}(P, M)$: This follows from the fact that the inference rules for the single-step semantics are a subset of the inference rules for the multi-step semantics.

□

In the next section we will present a formal method for reasoning about Gamma programs.

2.2 Reasoning about Gamma Programs

In this section we briefly introduce a method for reasoning about Gamma programs that is inspired on the UNITY logic [23] and its application to multiset transformer programming as first described in [79]. This method complements the methods proposed in [12] in that it is suitable for the a-posteriori verification of programs. Furthermore, it enables us to establish properties of Gamma programs that we can exploit in later stages of development where we concentrate on refinement of coordination strategies for Gamma programs.

We introduce a small repertoire of basic properties that suffices for the applications in this thesis. It is straightforward to extend this repertoire with other properties (such as appear in UNITY [23] or other temporal logics).

We write quantified predicates on multisets in the following way.

$$[\![\textit{quantifier variable-list} : \textit{range-expression} : \textit{boolean-expression}]\!] \textit{multiset}$$

The variables that occur in the *variable-list* range over all values in the *range-expression* that are present in a given multiset.

$$\begin{array}{lcl} [\![\forall \overline{x} : ran(\overline{x}) : p]\!]M & \Leftrightarrow & \forall \overline{v} : \overline{v} \subseteq M \wedge ran(\overline{v}) : p[\overline{x} := \overline{v}] \\ [\![\exists \overline{x} : ran(\overline{x}) : p]\!]M & \Leftrightarrow & \exists \overline{v} : \overline{v} \subseteq M \wedge ran(\overline{v}) : p[\overline{x} := \overline{v}] \end{array}$$

For example, $[\![\forall s, i, x_i : (\mathcal{X}, s, i, x_i) : s \geq 0]\!]M$ should be read as: 'for all values s, i, x_i such that there is a tuple $(\mathcal{X}, s, i, x_i)$ in multiset M, holds that s is greater than or equal to zero".

Following [23] we also use quantified expressions (over multisets) where a binary, associative and commutative operator with a unit element is used instead of a quantifier[2].

$$[\![\textit{ operator variable-list} : \textit{range-expression} : \textit{numerical-expression}]\!] \textit{ multiset}$$

For example, $[\![+ t, i, z : (\mathcal{Z}, t, i, z) : t]\!]M$ denotes the sum of all values t for which there is a tuple $(\mathcal{Z}, t, i, z)$ for some t, i and i in multiset M. If the range of the quantification is empty, then the value of the expression is the unit element of the operator. The unit elements of min, max, $+$, $*$ are ∞, $-\infty$, 0 and 1 respectively.

In addition, we use the symbol '#' as a counting quantifier (as introduced by [61]). Formally,

$$[\![\# x : p]\!]M \quad = \quad \Sigma_{a \in A} f(a) \text{ where } f(a) = \begin{cases} M(a) & \text{if } p[x := a] \\ 0 & \text{otherwise} \end{cases}$$

For example, $[\![(\# s, i, x : (\mathcal{X}, s, i, x)) : \textit{true}]\!]M$ denotes the number of tuples of the form $(\mathcal{X}, s, i, x)$ in the multiset M.

In contrast to [23], we define the properties of our logic in terms of the formal operational semantics of programs (Figure 2.3). Let q, q' be quantified predicates on multisets,

[2]Although this notation is a debatable deviation from the mathematical convention, it constitutes a uniform notation and avoids long subscripts (especially with $\sum$ or $\prod$) which regularly occur when working with tuples rather than numbers.

let M_i, M' etc. denote multisets. Let $\langle P, M_0\rangle$ be the initial configuration of some program P.

- *initially* q iff $[\![q]\!]M_0$
- q *unless* q' iff

  $$(\forall P', P'', M', M'' : \langle P, M_0\rangle \leadsto^* \langle P', M'\rangle \leadsto \langle P'', M''\rangle : ([\![q \wedge \neg q']\!]M' \Rightarrow [\![q \vee q']\!]M'')$$

  From an operational point of view, q *unless* q' means that if q holds at some point of the computation, and q' does not, then after the next transition, either q continues to hold or q' starts to hold.
- *stable* q iff q *unless false*
  A predicate q is stable, if, once predicate q holds at some point of the computation, it will continue to hold. However, q may never start to hold.
- *invariant* q iff *initially* $q \wedge$ *stable* q
  A invariant q is a stable predicate that holds throughout the computation.

In addition to these, we introduce the termination condition, denoted $\dagger P$, that characterizes the final state(s) of a (simple) program. Enabledness of a (simple) program, which is dual to termination, is denoted $\natural P$.

$$\begin{array}{lcl} [\![\natural(r_1 + \ldots + r_n)]\!]M & \Leftrightarrow & \exists i : 1 \le i \le n : (\exists \overline{v} : \overline{v} \subseteq M : b_i[\overline{x} := \overline{v}]) \\ [\![\dagger(r_1 + \ldots + r_n)]\!]M & \Leftrightarrow & \forall i : 1 \le i \le n : (\forall \overline{v} : \overline{v} \subseteq M : \neg b_i[\overline{x} := \overline{v}]) \end{array}$$

The termination condition of a program can be derived in a syntactical manner by negating the conditions of the rewrite rules that constitute the program.

Lemma 2.2.1 *Let $P = r_1 + \ldots + r_n$ be a simple program.*
If $\langle P, M\rangle\surd$, then $\forall i : 1 \le i \le n : [\![\dagger r_i]\!]M$

Proof By transition induction using the semantics from Figure 2.3 follows that $\langle P, M\rangle\surd \Leftrightarrow (\forall i : 1 \le i \le n : \langle r_i, M\rangle\surd)$. The result follows from $\langle r_i, M\rangle\surd \Leftrightarrow [\![\dagger r_i]\!]M$. □

A specification of the desired output of a program is called that program's postcondition. The postcondition of Gamma programs can be specified using the quantified predicates introduced above.

The correctness of a Gamma program can be established by showing that it satisfies a number of properties which together imply the postcondition. A good start for deducing properties of the output of a Gamma program is by calculating its termination condition. In addition to the termination condition, it may be necessary to find a suitable collection of invariants such that their conjunction implies the postcondition.

In the next section we will briefly illustrate this method by proving the correctness of the Gamma program *swap* for sorting from Example 2.1.1.

Correctness of the Sorting Program

We assume the input to the sorting program *swap* is a sequence $\overline{a_0} = \langle\, a_1, \ldots, a_n \,\rangle$. The sorting program is correct if it produces a rearrangement of the elements of the sequence in nondecreasing order.

Definition 2.2.2 *A sequence* $\bar{l} = \langle\, l_1, \ldots, l_n \,\rangle$ *is sorted iff*

$$\forall i, j : 1 \leq i, j \leq n : i < j \Rightarrow l_i \leq l_j \tag{2.2}$$

A pair of elements from the sequence which violates (2.2), is called an *inversion.* Hence, the sorted sequence is characterized by having no inversion.

The initial sequence $\overline{a}$ is represented by the multiset $M_0 = \{(a_i, i) \mid 1 \leq i \leq n\}$. The Gamma program for sorting consists of the single rewrite rule *swap* which exchanges two elements that form an inversion.

$$swap \mathrel{\widehat{=}} (i, x), (j, y) \mapsto (i, y), (j, x) \Leftarrow x > y \ \wedge\ i < j \tag{2.3}$$

Generally, the postcondition of a Gamma program falls into two parts: an existential part and a universal part . The existential part states that certain elements are present in the multiset and the universal part states that these elements are a solution to the problem.

To formally express the postcondition for the sorting program we introduce the following auxiliary definition.

Definition 2.2.3 *Let* $\bar{l}$ *be a sequence and let* k *be some value. Then* $\bar{l} \downarrow k$ *denotes the number of occurrences of* k *in* $\bar{l}$*.*

For an initial sequence $\overline{a}$, the postcondition can be expressed as follows

1. Existential: $\forall i : 1 \leq i \leq n : \#(i, x) = 1$

2. Universal:

   (a) $\forall i : 1 \leq i \leq n : \overline{a} \downarrow a_i = (\#(j, x) : x = a_i)$

   (b) $\forall i, j, x, y : (i, x), (j, y) : i < j \Rightarrow x \leq y$

We will proceed according to the following strategy. First, we will calculate the termination predicate of the sorting program. Next, we examine which properties need to hold in addition to the termination predicate such that the postcondition is met and attempt to prove one or more invariants which imply these additional properties. Finally, we prove that the program terminates.

Hence, for the sorting program, we first calculate the termination predicate. The termination predicate $\dagger$*swap* implies condition 2(b) of the postcondition. Next, we show that the remaining properties 1 and 2(a) are invariants of the sorting program *swap*.

Lemma 2.2.4 *invariant* $\forall i : 1 \leq i \leq n : \overline{a} \downarrow a_i = (\#(j, x) : x = a_i)$

Proof

- *initially* : follows from the definition of M_0.

- *stable* : We show that the property is preserved by every possible individual execution of *swap*. Assume the property holds in M and $\langle swap, M\rangle \overset{\sigma}{\rightsquigarrow}_1 \langle swap, M'\rangle$. Hence $\sigma = \{(i, x), (j, y)\}/\{(j, x), (i, y)\}$ where $i < j$ and $x > y$. From the fact that σ inserts the same values x and y as it removes, follows that the property continues to hold.

□

Lemma 2.2.5 *invariant* $\forall i : 1 \leq i \leq n : \#(i, x) = 1$

Proof Analogous to Lemma 2.2.4. □

We finish by showing that the program *swap* terminates. To this end, we define a metric I that maps a multiset M onto the number of inversions in (the sequence represented by) M

$$I(M) = (+i, j, x, y : (i, x), (j, y) \in M : i > j \wedge x < y : 1)$$

Because the initial sequence is finite, the number of inversions is finite. Furthermore, $\langle swap, M\rangle \overset{\sigma}{\rightsquigarrow}_1 \langle swap, M'\rangle$ implies $I(M') < I(M)$. The number of inversions is bounded from below because there can be no fewer than zero inversions. We conclude that the program terminates.

2.3 Concluding Remarks

In this chapter we have presented the Gamma model and provided it with a formal semantics in terms of a labelled multi-step transition system. This semantics will be used in the next chapter for defining the semantics of a coordination language for Gamma.

Based on the semantics of Gamma we have defined a logic for reasoning about programs. We illustrated a method for using this logic by showing how it can be used to prove the correctness of a sorting program. A more elaborate example of the application of the logic can be found in Chapter 7 where we use it to prove the correctness of a Gamma program for solving triangular systems of linear equations.