

Formal Semantics
of the
CHART Transformation Language

Maarten de Mol, Arend Rensink

M.J.deMol@utwente.nl, rensink@cs.utwente.nl

University of Twente, Netherlands

6th December, 2011

Contents

1	Introduction	1
2	Preliminaries	3
2.1	Constructors	3
2.2	Lists	3
2.3	Partial functions	4
3	Graphs and type graphs	5
3.1	Basic types and basic values	5
3.2	Type graphs	6
3.3	Types and values	7
3.4	Graphs	9
4	Transformations	13
4.1	Symbols	13
4.2	Operations	13
4.3	Expressions	14
4.4	Match statements	15
4.5	Update statements	15
4.6	Sequence statements	16
4.7	Rule systems	17
5	Semantics (matching, updating)	19
5.1	Context	19
5.2	Apply operation	20
5.3	Evaluate expression	22
5.4	Matching (and predicates)	22
5.5	Updating	24
5.5.1	Processing ‘let’ block	25
5.5.2	Pre-processing ‘set’ block	25
5.5.3	Simultaneous application of graph updates	26
5.5.4	Executing update block as a whole	27
6	Semantics (sequencing, rule systems)	29
6.1	Control automaton	29
6.2	Building the control automaton	31
6.3	Dynamic behavior	33
6.4	System automaton	35

Chapter 1

Introduction

This document describes the formal semantics of CHART, which is a custom transformation language developed for the RDT in the CHARTER project. The purpose of the semantics is to unambiguously determine, on the mathematical level, what the output is when a given transformation is applied to a given input graph. The semantics allows desirable properties of the transformation, such as preservation of semantics or termination, to be expressed formally, and subsequently allows criteria to be established which ensure that these properties actually hold in practice.

This document presents the semantics only, and is written completely on a theoretical, mathematical level. Although no advanced concepts are used, and all definitions are also explained informally, a basic understanding of the core concepts of formal mathematics and logic is still recommended for reading this deliverable. Moreover, knowledge about the RDT and its transformation language CHART is also required.

The remainder of this document is structured as follows. Chapter 2 introduces several preliminary notations that are used throughout this document. Chapters 3 and 4 define the mathematical data structures for graphs (and type graphs) and transformations, respectively. Chapters 5 and 6, finally, describe the behavior of transformations, in terms of mathematical functions that operate on the formalized data structures.

Chapter 2

Preliminaries

Our formal framework is mainly based on basic set theory, but we also use special notations for constructors, lists, and (partial) functions.

2.1 Constructors

A constructor is a special symbol that is written in sans-serif. It can have arguments, and an application is written without brackets, for instance as ‘cons $a b$ ’. Semantically, one can think of such an application as an implicit tuple, i.e. ‘(cons, a, b)’.

2.2 Lists

A list is a dedicated notation for an ordered sequence, which will be used frequently in the formalization. We use $\ell(A)$ (analogously to $\wp(A)$) to denote the set of all possible lists over A , and $\langle a_1, \dots, a_n \rangle$ (analogously to $\{a_1, \dots, a_n\}$) to denote lists themselves.

Definition 2.2.1: (*lists*)

The set of lists over elements of an arbitrary set A will be denoted by $\ell(A)$. It is built inductively, using ‘ $\langle \rangle$ ’ for the empty list and ‘ $\langle a : as \rangle$ ’ for the list constructor, as follows:

$$\ell(A) = \{\langle \rangle\} \cup \{\langle a : as \rangle \mid a \in A, as \in \ell(A)\}$$

Notation 2.2.2: (*list notation*)

Let $\langle a_1 \dots a_n \rangle$ abbreviate $\langle a_1 : \langle a_2 : \dots \langle a_n : \langle \rangle \rangle \rangle \rangle$.

If $n = 0$, this is the empty list $\langle \rangle$.

Definition 2.2.3: (*list operations*)

The following standard operations are defined for lists:

- *element of:* $x \in \langle a_1 \dots a_n \rangle \Leftrightarrow \exists 1 \leq i \leq n [a_i = x]$
- *size:* $|\langle a_1 \dots a_n \rangle| = n$
- *concatenation:* $\langle \rangle \oplus B = B$ and $\langle a : A \rangle \oplus B = \langle a : A \oplus B \rangle$
- *convert to set:* $\bar{V} = \{v \mid v \in V\}$

Definition 2.2.4: (*flatten list of lists*)

For all sets A , the function $flt : \ell(\ell(A)) \rightarrow \ell(A)$ is defined by:

$$\begin{aligned} flt(\langle \rangle) &= \langle \rangle \\ flt(\langle x:y \rangle) &= x \oplus flt(y) \end{aligned}$$

2.3 Partial functions

We explicitly distinguish *partial* functions from total ones by using \hookrightarrow instead of \rightarrow . For a partial function $f : A \hookrightarrow B$, $Dom(f) \subseteq A$ holds, and for $f : A \rightarrow B$, $Dom(f) = A$ holds. Consequently, $(A \rightarrow B) \subseteq (A \hookrightarrow B)$. Both kinds of functions will often be interpreted as sets of (a, b) pairs.

We also introduce explicit notation for substitution on functions, which updates a function with new (source,target) pairs.

Notation 2.3.1: (*function updating*)

For all sets A and B , all functions $f : A \hookrightarrow B$, and all elements $a \in A$ and $b \in B$, let $f[a \mapsto b]$ be defined by:

$$f[a \mapsto b](a') = \begin{cases} b & \text{if } a = a' \\ f(a) & \text{otherwise} \end{cases}$$

Notation 2.3.2: (*function updating, lists*)

For all sets A and B , all functions $f : A \hookrightarrow B$, and all lists $as \in \ell(A)$ and $bs \in \ell(B)$, let $f[as \mapsto bs]$ be defined by:

$$f[as \mapsto bs] = \begin{cases} f[a' \mapsto b'][as' \mapsto bs'] & \text{if } as = \langle a' : as' \rangle \wedge bs = \langle b' : bs' \rangle \\ f & \text{otherwise} \end{cases}$$

Notation 2.3.3: (*function space*)

For all sets A and B , let A^B denote the set of partial functions from A to B :

$$A^B = A \hookrightarrow B$$

Chapter 3

Graphs and type graphs

In the following sections, we will formalize the objects that are transformed by a CHART transformation, which are rooted and connected graphs. In Section 3.1, basic types and basic values are introduced first. In Section 3.2, the *type graphs* are defined against which graphs will be typed. In Section 3.3, basic types and values are extended to arbitrary types and values. In Section 3.4, the graphs themselves are finally defined.

3.1 Basic types and basic values

Every graph can store elementary basic values. In our case, we allow booleans, characters, real numbers, whole numbers, and strings. We assume an approximated algebra to be available for real numbers. For characters, we simply assume an abstract set of representing values.

Definition 3.1.1: (*algebra for \mathbb{R}*)

Assume that \mathbb{R}_{\approx} is an algebra that approximates \mathbb{R} . All the standard mathematical operations are assumed to be available for \mathbb{R}_{\approx} . Furthermore, \mathbb{R}_{\approx} is assumed to be countable (or even finite), and is at least assumed to behave in accordance to the IEEE 754 standard.

Definition 3.1.2: (*booleans*)

The set of boolean values *Bool* is defined by:

$$Bool = \{\text{true}, \text{false}\}$$

Assumption 3.1.3: (*characters*)

Assume that *Char* represents the set of allowed character values.

Definition 3.1.4: (*strings*)

The set of string values *String* is defined by:

$$String = \ell(Char)$$

Definition 3.1.5: (*basic values*)

The set of basic values \mathcal{V}_b is defined by:

$$\begin{aligned} \mathcal{V}_b = & \{\text{bool } v \mid v \in Bool\} \\ & \cup \{\text{char } v \mid v \in Char\} \\ & \cup \{\text{float } r \mid r \in \mathbb{R}_{\approx}\} \\ & \cup \{\text{int } n \mid n \in \mathbb{Z}\} \\ & \cup \{\text{string } v \mid v \in String\} \end{aligned}$$

A basic value is typed by a basic type, which is defined straightforwardly. Note that throughout this document, we will denote *values* with the letter \mathcal{V} , and *types* with the letter \mathcal{T} .

Definition 3.1.6: (*basic types*)

The set of basic types \mathcal{T}_b is defined by:

$$\mathcal{T}_b = \{\text{bool, char, float, int, string}\}$$

Definition 3.1.7: (*typing of basic values*)

The typing function $\text{type}_b : \mathcal{V}_b \rightarrow \mathcal{T}_b$ is defined by:

$$\text{type}_b(\text{bool } v) = \text{bool}$$

$$\text{type}_b(\text{char } v) = \text{char}$$

$$\text{type}_b(\text{float } r) = \text{float}$$

$$\text{type}_b(\text{int } n) = \text{int}$$

$$\text{type}_b(\text{string } v) = \text{string}$$

Finally, we introduce $\beta(P)$ as a notation for explicitly converting the logical statement P into a boolean value.

Notation 3.1.8: (*convert to boolean*)

Let $\beta(P)$ be defined by:

$$\beta(P) = \begin{cases} \text{true} & \text{if } P \\ \text{false} & \text{if } \neg P \end{cases}$$

3.2 Type graphs

A type graph describes the allowed structure of a graph. From a modeling point of view, it corresponds to a *meta model*. In our formalization, a type graph consists of node types and field types (a field is a unified view on attributes and edges), for which the following properties are defined:

- A field type *connects* a source node type to either a target node type (binary edge) or a basic type (attribute).
- A node type can be a *subtype* of another node type. A subtype inherits all the field types from its supertype. Multiple inheritance is allowed in our framework.
- A node type can be *abstract*. Nodes of an abstract node type may not appear in an instance graph.
- Each field type has a minimum and a maximum *multiplicity*. In an instance graph, a field can connect a single source to multiple targets, but the exact number must always be in the multiplicity range of its field type.
- A field type can be *ordered*, which means that its values in an instance graph are lists. Values of an unordered field type are sets.

The type graph defines the set of available node types. We require such a locally defined set to be a subset of a *globally* defined set of all possible node types. This allows future structures to refer to node types, without explicitly requiring a type graph to be in the context. Determining the meaning of such a structure, however, does require the type graph.

We assume global sets to be available for node and field types, as follows:

Assumption 3.2.1: (*global set of node types*)

Assume that a global set of node types is available by means of the set \mathcal{T}_n .

Assumption 3.2.2: (*global set of field types*)

Assume that a global set of field types is available by means of the set \mathcal{T}_f .

Using these global node and field types, a type graph structure can be defined as follows:

Definition 3.2.3: (*type graphs*)

A type graph is a structure $(N, F, src, tgt, abs, \leq_t, min, max, ord)$, in which:

- $N \subseteq \mathcal{T}_n$ is the set of defined node types;
- $F \subseteq \mathcal{T}_f$ is the set of defined field types;
- $src : F \rightarrow N$ gives the source (node type) of a field type;
- $tgt : F \rightarrow N \cup \mathcal{T}_b$ gives the target (node or basic type) of a field type;
- $abs \subseteq N$ is the subset of node types that are abstract;
- $\leq_t \subseteq N \times N$ is the subtyping relation on node types, which must be a partial order;
- $min : F \rightarrow \mathbb{N}$ and $max : F \rightarrow \mathbb{N} \cup \{\text{many}\}$ are the multiplicity functions for field types, for minimum and maximum values respectively;
- $ord \subseteq F$ is the subset of field types that are ordered.

The universe of type graphs will be denoted by \mathcal{T}_G .

If $T \in \mathcal{T}_G$ is a type graph, then $src_T, tgt_T, abs_T, \leq_T, min_T, max_T$ and ord_T abbreviate the $src, tgt, abs, \leq_t, min, max$ and ord elements of T , respectively.

Properties that are not yet modeled in our semantics, but are available in CHART, are edge opposites and containment. It is future work to extend our semantics with these concepts as well.

3.3 Types and values

The basic unit of information that is stored in an instance graph will be called a *value*. A value is the result of navigating over a field type from a given source node. Depending on the field type, this result can either be:

- A basic value, if the maximum multiplicity of the field type is 1 and its target type is a basic type.
- A graph node, if the maximum multiplicity of the field type is 1 and its target type is a node type.
- A collection of one of the above, if the maximum multiplicity of the field type is greater than 1.
 - The collection is a *list* if the field type is ordered.
 - The collection is a *set* if the field is not ordered.

Note that in our framework, collections are first class values that can also be manipulated as a whole. This increases the expressiveness of CHART, and can be justified on the theoretical level by using *hyperedges* instead of binary ones.

Values refer to graph nodes, which in turn are defined by graphs, which have not yet been introduced. We will use the same mechanism as for node and field types, and define a global set of available nodes first. A node will be represented by a tuple of a node type and a natural number. This allows the type of a node to be determined statically, and allows fresh nodes to be created when needed.

Definition 3.3.1: (*global graph nodes*)

The set of global graph nodes \mathcal{N} is defined by:

$$\mathcal{N} = \{\text{node } i t \mid i \in \mathbb{N}, t \in \mathcal{T}_n\}$$

Definition 3.3.2: (*obtain node type*)

The node type of a node can be retrieved by the function $\text{type}_n : \mathcal{N} \rightarrow \mathcal{T}_n$, which is straightforwardly defined by $\text{type}_n(\text{node } i t) = t$.

Using the set of nodes, the set of values can be formalized as follows:

Definition 3.3.3: (*values*)

The set of values \mathcal{V} is defined by:

$$\begin{aligned} \mathcal{V} = & \{\perp\} \cup \mathcal{V}_b \cup \mathcal{N} \\ & \cup \{\text{list } V \mid V \in \ell(\mathcal{V})\} \\ & \cup \{\text{set } V \mid V \in \wp(\mathcal{V})\} \end{aligned}$$

Definition 3.3.4: (*lift $|\cdot|$ to values*)

The function $|\cdot| : \mathcal{V} \rightarrow \mathbb{N}$ is defined by:

$$|v| = \begin{cases} |V| & \text{if } v = \text{list } V \\ |V| & \text{if } v = \text{set } V \\ 1 & \text{otherwise} \end{cases}$$

Definition 3.3.5: (*lift \in to values*)

The relation $\in \subseteq \mathcal{V} \times \mathcal{V}$ is defined by:

$$\begin{aligned} v_1 \in v_2 \Leftrightarrow & v_2 = \text{list } V \wedge v_1 \in V \\ & \vee v_2 = \text{set } V \wedge v_1 \in V \end{aligned}$$

Definition 3.3.6: (*nodes in a value*)

The function $\text{nodes} : \mathcal{V} \rightarrow \wp(\mathcal{N})$ is defined by:

$$\text{nodes}(v) = \begin{cases} \emptyset & \text{if } v = \perp \vee v \in \mathcal{V}_b \\ \{v\} & \text{if } v \in \mathcal{N} \\ \cup_{v' \in V} [\text{nodes}(v')] & \text{if } v = \text{list } V \\ \cup_{v' \in V} [\text{nodes}(v')] & \text{if } v = \text{set } V \end{cases}$$

The set of value types is defined analogously to the set of values. It consists of basic types, node types, and special list and set types for collections.

Definition 3.3.7: (*types*)

The set of types \mathcal{T} is defined by:

$$\begin{aligned} \mathcal{T} = & \mathcal{T}_b \cup \mathcal{T}_n \\ & \cup \{\text{list } t \mid t \in \mathcal{T}\} \\ & \cup \{\text{set } t \mid t \in \mathcal{T}\} \end{aligned}$$

With $v ::_T t$ we will denote that v is a valid value for the type t relative to the type graph T . Our typing follows the subtyping relation on node types, and the error value \perp is a valid value of any type.

Definition 3.3.8: (*valid values for a given type*)

For any type graph $T \in \mathcal{T}_G$, the typing relation $::_T \subseteq \mathcal{V} \times \mathcal{T}$ is defined by:

$$\begin{aligned} v ::_T t &\Leftrightarrow v \in \mathcal{V}_b \wedge t \in \mathcal{T}_b \wedge \text{type}_b(v) = t \\ &\vee v \in \mathcal{N} \wedge t \in \mathcal{T}_n \wedge \text{type}_n(v) \leq_T t \\ &\vee \exists V \in \ell(\mathcal{V}) \exists t' \in \mathcal{T} [v = \text{list } V \wedge t = \text{list } t' \wedge \forall v' \in V [v' ::_T t']] \\ &\vee \exists V \in \rho(\mathcal{V}) \exists t' \in \mathcal{T} [v = \text{set } V \wedge t = \text{set } t' \wedge \forall v' \in V [v' ::_T t']] \\ &\vee v = \perp \end{aligned}$$

Each field type in a given type graph can be associated with a unique target value type, which is determined on the basis of its target (*tgt*), its multiplicity (*min* and *max*), and its orderedness (*ord*). This target value type can be obtained with the *type_f* function.

Definition 3.3.9: (*value type of a field type*)

The function $\text{type}_f : \mathcal{T}_f \times \mathcal{T}_G \rightarrow \mathcal{T}$ is defined by:

$$\text{type}_f(f, T) = \begin{cases} \text{list } \text{tgt}(f) & \text{if } \text{max}(f) > 1 \wedge \text{ord}(f) \\ \text{set } \text{tgt}(f) & \text{if } \text{max}(f) > 1 \wedge \neg \text{ord}(f) \\ \text{tgt}(f) & \text{otherwise} \end{cases}$$

3.4 Graphs

A graph defines a set of nodes, and stores values for the fields of those nodes. In addition, our graphs are *rooted*. This can be formalized straightforwardly:

Definition 3.4.1: (*graphs*)

A graph is a structure (N, r, F) , in which:

- $N \subseteq \mathcal{N}$ is the set of nodes in the graph;
- $r \in N$ is the designated root node of the graph;
- $F : N \times \mathcal{T}_f \leftrightarrow \mathcal{V}$ are the field values in the graph.

The universe of graphs will be denoted with \mathcal{G} . For each graph $G = (N, r, F)$, G_N denotes N , G_r denotes r , and G_F denotes F .

This formalization of graphs refers to the global set of nodes \mathcal{N} and the global set of field types \mathcal{T}_f , and is therefore fully independent of a particular type graph. To express welltypedness, we explicitly introduce the following two relations:

- A graph is *welltyped* with respect to a type graph if the field function respects the source and target types of the field types.
- A welltyped graph is *wellformed* if also the number of elements in each collection value are within the multiplicity range of the corresponding field type, and no abstract node types appear in the graph.

These typing conditions are formalized as follows:

Definition 3.4.2: (*welltypedness of graphs*)

The relation $\text{welltyped} \subseteq \mathcal{G} \times \mathcal{T}_G$ is defined by:

$$\begin{aligned} \text{welltyped}(G, T) &\Leftrightarrow \forall (n, f, v) \in G_F [\text{type}_n(n) \leq_T \text{src}_T(f) \\ &\quad \wedge v ::_T \text{type}_f(f, T) \\ &\quad \wedge \text{nodes}(v) \subseteq G_N] \end{aligned}$$

Definition 3.4.3: (*wellformedness of graphs*)

The relation $wellformed \subseteq \mathcal{G} \times \mathcal{T}_{\mathcal{G}}$ is defined by:

$$wellformed(G, T) \Leftrightarrow \forall n \in G_N [\neg abs_T(type_n(n))] \\ \wedge \forall (n, f, v) \in G_F [\min_T(f) \leq |v| \\ \wedge (max_T(f) = \text{many} \vee |v| \leq max_T(f))]$$

Our graphs are not only rooted, but also *connected*. A node is only considered to be element of a graph if it is reachable from the root. In our formalization, the graph does not keep track of its reachable nodes explicitly. Instead, the node set G_N also stores unreachable nodes, and an explicit analysis is required to determine if a node is reachable.

Definition 3.4.4: (*step*)

For a given graph G , the step relation $\rightarrow_G \subseteq N \times N$ is defined by:

$$n_1 \rightarrow_G n_2 \Leftrightarrow \exists f \in \mathcal{T}_f [(n_1, f, n_2) \in G_F] \\ \vee \exists f \in \mathcal{T}_f \exists V \in \ell(\mathcal{V}) [(n_1, f, \text{list } V) \in G_F \wedge n_2 \in V] \\ \vee \exists f \in \mathcal{T}_f \exists V \in \emptyset(\mathcal{V}) [(n_1, f, \text{set } V) \in G_F \wedge n_2 \in V]$$

Let \rightarrow_G^* be the reflexive, transitive closure of \rightarrow_G .

Definition 3.4.5: (*reachable nodes*)

The function $reach : \mathcal{G} \rightarrow \wp(\mathcal{N})$ is defined by:

$$reach(G) = \{n \mid n \in G_N \mid G_r \rightarrow_G^* n\}$$

Traversing a field type from a give source node is formalized by means of the *get* function. If the graph does not contain a value for the field, then the function returns \perp . This can happen when the field type is not defined for the source node type, but also when the field has not yet been initialized. A field with maximum multiplicity greater than 1 is always considered to be initialized, however, and will return the empty collection instead.

Definition 3.4.6: (*get field value*)

The function $get : \mathcal{N} \times \mathcal{T}_f \times \mathcal{G} \rightarrow \mathcal{V}$ is defined by:

$$get(n, f, G) = \begin{cases} G_F(n, f) & \mathbf{if} (n, f) \in Dom(G_F) \\ \text{list } \langle \rangle & \mathbf{if} (n, f) \notin Dom(G_F) \wedge n \leq_t src(f) \\ & \wedge max(f) > 1 \wedge ord(f) \\ \text{set } \emptyset & \mathbf{if} (n, f) \notin Dom(G_F) \wedge n \leq_t src(f) \\ & \wedge max(f) > 1 \wedge \neg ord(f) \\ \perp & \mathbf{otherwise} \end{cases}$$

The function *set* changes (possibly many) field values in the graph:

Definition 3.4.7: (*set field values*)

The function $set : (\mathcal{N} \times \mathcal{T}_f \hookrightarrow \mathcal{V}) \times \mathcal{G} \rightarrow \mathcal{G}$ is defined by:

$$set(V, G) = (G_N, G_r, F) \\ \mathbf{where} F(n, f) = \begin{cases} F(n, f) & \mathbf{if} (n, f) \in Dom(F) \\ G_F(n, f) & \mathbf{otherwise} \end{cases}$$

For convenience, we abbreviate $set(\{(n, f, v)\}, G)$ to $set(n, f, v, G)$.

The third, and final, graph operation that needs to be available is the creation of a new node of a specific type. This is accomplished by the function *new*:

Definition 3.4.8: (*create new initialized node in the graph*)

The function $new : \mathcal{T}_n \times \mathcal{G} \rightarrow \mathcal{N} \times \mathcal{G}$ is defined by:

$$new(t, G) = (n, (G_N \cup \{n\}, G_r, G_F))$$

where $n = \text{node}(1 + \prod\{i \mid (\text{node } i t) \in G_N\}) t$

Chapter 4

Transformations

In this section, a bottom-up formalization of CHART transformations will be given. Sections 4.1, 4.2 and 4.3 describe the smallest building blocks, which are symbols, operations, and expressions respectively. Sections 4.4, 4.5 and 4.6 build statements out of these basic components, for match, update, and sequence blocks respectively. Section 4.7, finally, defines rules and rule systems in terms of statements and expressions.

4.1 Symbols

CHART transformations consist of rules (and predicates, which are a special kind of rules). A rule is defined by associating a unique rule *symbol* with a rule body. The symbol can then be used to refer to the rule, for instance in expressions and statements. By using rule symbols in rule bodies, recursion (and mutual recursion) can be expressed.

In the formalization, we simply assume the existence of two abstract sets of symbols, for identifying rules and predicates respectively:

Assumption 4.1.1: (*rule symbols*)

Assume that \mathcal{R} is the set of allowed rule symbols.

Assumption 4.1.2: (*predicate symbols*)

Assume that \mathcal{P} is the set of allowed predicate symbols.

4.2 Operations

The CHART language makes the following operations, with which values can be manipulated in transformations, available:

- Logical operations: negation (not), conjunction (and).
- Arithmetic operations: addition (plus), subtraction (minus), multiplication (times), division (div), modulo (mod).
- Comparison operations: equality (eq) and lesser than (lt).
- Selection operations: select at index (sel-at), select up to index (sel-upto), select from index (sel-from), select between two indices (between).
- Collection operations: set union (plus), set subtraction (minus), list concatenation (plus), get size (size), check membership (el-of).

- Graph operations: traverse given field from a node (get-field).
- Type operations: check if a node is an instance of a given node type (inst-of).

Operations that are not mentioned above are disjunction, greater than, lesser or equal than, greater or equal than, and index of. These are all part of the CHART language, but can be derived from the operations above. Therefore, they are not modeled explicitly in the formalization.

Below, symbols are introduced for the available operations. The symbols are separated on the basis of their arity. The behavior of the operations is formalized later, in Chapter 5.

Definition 4.2.1: (*operations with arity 1*)

The set \mathcal{O}_1 of allowed operations with arity 1 is defined by:

$$\begin{aligned} \mathcal{O}_1 = & \{\text{not, size}\} \\ & \cup \{\text{inst-of } t \mid t \in \mathcal{T}_n\} \\ & \cup \{\text{get-field } f \mid f \in \mathcal{T}_f\} \end{aligned}$$

Definition 4.2.2: (*operations with arity 2*)

The set \mathcal{O}_2 of allowed operations with arity 2 is defined by:

$$\begin{aligned} \mathcal{O}_2 = & \{\text{and, div, el-of, eq, lt, minus, mod, plus, times}\} \\ & \cup \{\text{sel-at, sel-from, sel-upto}\} \end{aligned}$$

Definition 4.2.3: (*operations with arity 3*)

The set \mathcal{O}_3 of allowed operations with arity 3 is defined by:

$$\mathcal{O}_3 = \{\text{between}\}$$

Definition 4.2.4: (*all operations*)

The set \mathcal{O} of all operations is defined by:

$$\mathcal{O} = \mathcal{O}_1 \cup \mathcal{O}_2 \cup \mathcal{O}_3$$

4.3 Expressions

Expressions are the basic computational units that can be written down in a CHART transformation. They can be used at any point in a rule, and compute a value, possibly by inquiring the current graph. However, the graph can never be changed by the computation. An expression can be one of the following:

- A variable (which was bound to a value in the context).
- A computed value.
- The application of an operation on expression arguments.
- The application of a predicate on expression arguments.
- A node set, which denotes the set of all (reachable) graph nodes that are of a specific type (or a subtype of it).

This can be formalized straightforwardly, as follows. Note that we use \mathcal{X} to denote the set of variables, as \mathcal{V} already denotes the set of values.

Definition 4.3.1: (*variables*)

The set of variables \mathcal{X} is defined by:

$$\mathcal{X} = \{\text{var } i t \mid i \in \mathbb{N}, t \in \mathcal{T}\}$$

Definition 4.3.2: (*obtain variable type*)

The type of a variable can be retrieved by the function $type_x : \mathcal{X} \rightarrow \mathcal{T}$, which is straightforwardly defined by $type_x(\text{var } i \ t) = t$.

Definition 4.3.3: (*expressions*)

The set of expressions \mathcal{E} is defined by:

$$\begin{aligned} \mathcal{E} = & \mathcal{X} \cup \mathcal{V} \\ & \cup \{\text{op } o \ E \mid o \in \mathcal{O}, E \in \ell(\mathcal{E})\} \\ & \cup \{\text{pred } p \ E \mid p \in \mathcal{P}, E \in \ell(\mathcal{E})\} \\ & \cup \{\text{nodeset } t \mid t \in \mathcal{T}_n\} \end{aligned}$$

4.4 Match statements

The match block of a CHART transformation searches for a specific pattern in the instance graph. It consists of a list of match statements, which can be one of the following:

- A match variable, which specifies a pattern element to look for.
- A boolean expression, which specifies a condition that must hold for the pattern elements to look for.
- A forall statement, which specifies a condition that must hold for all elements of a collection.

This can be formalized straightforwardly, as follows:

Definition 4.4.1: (*match statement*)

The set of match statements \mathcal{S}_m is defined by:

$$\begin{aligned} \mathcal{S}_m = & \{\text{search } x \mid x \in \mathcal{X}\} \\ & \cup \{\text{check } e \mid e \in \mathcal{E}\} \\ & \cup \{\text{forall } x \ e \ S \mid x \in \mathcal{X}, e \in \mathcal{E}, S \in \ell(\mathcal{S}_m)\} \end{aligned}$$

4.5 Update statements

The update block of a CHART transformation specifies changes that must be applied to an instance graph. The block consists of a separate ‘let’ block, for creating nodes and assigning variables, and a separate ‘set’ block, for setting fields in the graph. The ‘let’ block will be executed sequentially, and always before the ‘set’ block. The field updates in the ‘set’ block will be executed simultaneously.

A ‘let’ block consists of a sequence of update ‘let’ statements. A ‘let’ statement can be one of the following:

- The assignment of an expression to a variable. This introduces an alias for the expression. Also, it allows the value of the expression (from before the update) to be referenced after the update block.
- The creation of a single node in the graph. The created node must always be assigned to a variable, which allows the node to be referenced in the subsequent components of the update block.

The created nodes are never initialized in the formalization. This is not a problem, because an initialized node creation can be mapped into an uninitialized one, as follows. Suppose

that a node n is created, and a field f must be initialized to e . This is equivalent to an uninitialized node creation, combined with an explicit field initialization $n.f = e$ in the ‘set’ block. Also, each other reference to $n.f$ in the ‘set’ block must be replaced by e , as it will be evaluated simultaneously with the initialization itself.

Definition 4.5.1: (*update let statements*)

The set $\mathcal{S}_{u.l}$ of update ‘let’ statements is defined by:

$$\mathcal{S}_{u.l} = \{\text{assign } x \ e \mid x \in \mathcal{X}, e \in \mathcal{E}\} \\ \cup \{\text{create } x \ t \mid x \in \mathcal{X}, t \in \mathcal{T}_n\}$$

A ‘set’ block consists of a sequence of update ‘set’ statements. The order of the statements does not matter, as they will be executed simultaneously. A ‘set’ statement can be one of the following:

- The assignment of an expression to a field of a node. This updates the field as a whole, and the old value is discarded. It is also allowed for collection types, in which case the new value is a new collection itself.
- The assignment of an expression to a *specific index* of a field of a node. This is only valid for list fields, and does not affect the other elements of the existing list value.
- An iteration of an argument update block over a collection. The argument block consists of both ‘let’ and ‘set’ statements. The ‘let’ statements are implicitly regarded as part of the overall ‘let’ block, and will be extracted by the operational semantics.

Definition 4.5.2: (*update set statements*)

The set $\mathcal{S}_{u.s}$ of update ‘set’ statements is defined by:

$$\mathcal{S}_{u.s} = \{\text{set } e_1 \ f \ e_2 \quad \mid e_1, e_2 \in \mathcal{E}, f \in \mathcal{T}_f\} \\ \cup \{\text{seti } e_1 \ f \ e_2 \ e_3 \quad \mid e_1, e_2, e_3 \in \mathcal{E}, f \in \mathcal{T}_f\} \\ \cup \{\text{foreach } x \ e \ S_1 \ S_2 \mid x \in \mathcal{X}, e \in \mathcal{E}, S_1 \in \ell(\mathcal{S}_{u.l}), S_2 \in \ell(\mathcal{S}_{u.s})\}$$

4.6 Sequence statements

The sequence block of a CHART transformation establishes flow of control. It consists of imperative statements, which are executed sequentially. It is the only block in which rule calls are allowed. The following kinds of sequence statements are available:

- An assignment of a value to a variable.
- An application of a rule on given arguments. The result of the application is stored in a list of local variables.
- An if statement, which chooses between two blocks based on a boolean expression.
- A try statement, which catches rule failure in a given block. If a rule failure is caught successfully, execution continues with the else block if it exists, and terminates the try statement with success otherwise.
- A foreach statement, which executes a block for all elements of a given collection.
- A repeat statement, which repeats a block until rule failure is caught in it. If rule failure is caught successfully, the repeat statement terminates with success.

Definition 4.6.1: (*sequence statements*)

The set \mathcal{S}_s of sequence statements is defined by:

$$\begin{aligned} \mathcal{S}_s = & \{ \text{assign } x \ e \mid x \in \mathcal{X}, e \in \mathcal{E} \} \\ & \cup \{ \text{apply } X \ r \ E \mid X \in \ell(\mathcal{X}), r \in \mathcal{R}, E \in \ell(\mathcal{E}) \} \\ & \cup \{ \text{if } e \ S_1 \ S_2 \mid e \in \mathcal{E}, S_1, S_2 \in \ell(\mathcal{S}_s) \} \\ & \cup \{ \text{try } S_1 \ S_2 \mid S_1, S_2 \in \ell(\mathcal{S}_s) \} \\ & \cup \{ \text{foreach } x \ e \ S \mid x \in \mathcal{X}, e \in \mathcal{E}, S \in \ell(\mathcal{S}_s) \} \\ & \cup \{ \text{repeat } S \mid S \in \ell(\mathcal{S}_s) \} \end{aligned}$$

4.7 Rule systems

A CHART rule system consists of a set of rules, a designated start rule, and a set of predicates:

- A rule is defined by a rule symbol, a list of input variables, a list of match statements (the match block), a list of update ‘let’ statements and a list of update ‘set’ statements (the update block), a list of sequence statements (the sequence block), and a list of return expressions.
- The start rule is the designated rule which begins the transformation as a whole. It is not allowed to have input parameters.
- A predicate is a special rule that only consists of input variables and a match block. It cannot have a side effect, and may therefore be called in an arbitrary expression (and thus also in match and update blocks). For this reason, predicates are distinguished syntactically from rules.

Definition 4.7.1: (*rule systems*)

A rule system RS is a structure $(R, P, \text{start}, \text{input}, \text{matchb}, \text{updateb}, \text{sequenceb}, \text{return})$, in which:

- $R \subseteq \mathcal{R}$ is the set of defined rule symbols;
- $P \subseteq \mathcal{P}$ is the set of defined predicate symbols;
- $\text{start} \in R$ is the designated start rule;
- $\text{input} : \mathcal{R} \cup \mathcal{P} \hookrightarrow \ell(\mathcal{X})$ associates symbols to input variables;
- the start rule has no input variables, i.e. $\text{input}(\text{start}) = \langle \rangle$;
- $\text{matchb} : \mathcal{R} \cup \mathcal{P} \hookrightarrow \ell(\mathcal{S}_m)$ associates symbols to match blocks;
- $\text{updateb} : \mathcal{R} \hookrightarrow \ell(\mathcal{S}_{u:l}) \times \ell(\mathcal{S}_{u:s})$ associates rule symbols to update blocks;
- $\text{sequenceb} : \mathcal{R} \hookrightarrow \ell(\mathcal{S}_s)$ associates rule symbols to sequence blocks;
- $\text{return} : \mathcal{R} \hookrightarrow \ell(\mathcal{E})$ associated rule symbols to return expressions;
- $\text{input}, \text{matchb}, \text{updateb}, \text{sequenceb}$ and return are defined for all local symbols; i.e.

$$\text{Dom}(\text{input}) = \text{Dom}(\text{matchb}) = R \cup P \text{ and}$$

$$\text{Dom}(\text{updateb}) = \text{Dom}(\text{sequenceb}) = \text{Dom}(\text{return}) = R.$$

The universe of rule systems will be denoted by \mathcal{RS} .

If $X \in \mathcal{RS}$ is a rule system, then $\text{rules}_X, \text{preds}_X, \text{start}_X, \text{input}_X, \text{matchb}_X, \text{updateb}_X, \text{sequenceb}_X$ and return_X abbreviate the $R, P, \text{start}, \text{matchb}, \text{updateb}, \text{sequenceb}$ and return elements of X , respectively.

Chapter 5

Semantics (matching, updating)

In the following subsections, the operational semantics (i.e. the behavior) is defined of *expressions*, *match blocks* and *update blocks*. These components will then be used in Chapter 6 to describe the behavior of a rule system as a whole.

- In Section 5.1, a convenient notation is introduced for referring to graphs, type graphs, and rule systems. These context structures are necessary input for nearly all semantic functions.
- In Section 5.2, the behavior of operations is defined, by means of a function that computes the effect of an operation on a list of input values.
- In Section 5.3, the behavior of expressions is defined, by means of a function that evaluates an expression to a value.
- In Section 5.4, the behavior of match blocks is defined, by means of a function that computes all possible matches of a match block.
- In Section 5.5, the behavior of update blocks is defined, by means of a function that computes the effect of an update block on an input graph.

Note that the behavior of sequence blocks is intertwined with the behavior of the rule system as a whole, because a sequence block is allowed to call other rules. Therefore, sequence blocks will be described as part of Chapter 6.

5.1 Context

The semantic functions that will be defined in the following sections require a (fixed) context in order to determine the behavior of expressions and statements. This context consists of a type graph, which is needed for subtyping, and a rule system, which is needed for applying rules and predicates.

To conveniently access this context, we introduce the concept of a *contextual graph*. This is simply a tuple of a normal graph, a type graph and a rule system, out of which the context structures can be extracted easily:

Definition 5.1.1: (*contextual graph*)

The set of contextual graphs \mathcal{G}_C is defined by:

$$\mathcal{G}_C = \mathcal{G} \times \mathcal{T}_G \times \mathcal{RS}$$

Definition 5.1.2: (*get type graph from contextual graph*)

The function $tg : \mathcal{G}_C \rightarrow \mathcal{T}_G$ is defined by:

$$tg(G, T, R) = T$$

Definition 5.1.3: (*get rule system from contextual graph*)

The function $rs : \mathcal{G}_C \rightarrow \mathcal{RS}$ is defined by:

$$rs(G, T, R) = R$$

Our semantic functions will operate on contextual graphs, instead of on normal ones. This makes the type graph and rule system available, using the tg and rs functions that are defined above. To use the graph component, we modify the graph functions of Section 3.4, as follows:

Definition 5.1.4: (*reach, contextual*)

The function $reach_C : \mathcal{G}_C \rightarrow \wp(\mathcal{N})$ is defined by:

$$reach_C(G, T, R) = reach(G)$$

Definition 5.1.5: (*get, contextual*)

The function $get_C : \mathcal{N} \times \mathcal{T}_f \times \mathcal{G}_C \rightarrow \mathcal{V}$ is defined by:

$$get_C(n, f, (G, T, R)) = get(n, f, G)$$

Definition 5.1.6: (*set, contextual*)

The function $set_C : (\mathcal{N} \times \mathcal{T}_f \hookrightarrow \mathcal{V}) \times \mathcal{G}_C \rightarrow \mathcal{G}_C$ is defined by:

$$set_C(F, (G, T, R)) = (set(F, G), T, R)$$

Definition 5.1.7: (*new, contextual*)

The function $new_C : \mathcal{T}_n \times \mathcal{G}_C \rightarrow \mathcal{N} \times \mathcal{G}_C$ is defined by:

$$new_C(t, (G, T, R)) = (n, (G', T, R)) \text{ if } new(t, G) = (n, G')$$

5.2 Apply operation

The behavior of an operation (see Definition 4.2.4) is determined by a function that transforms a list of values (the input) to a single result value (the output). If too few, too many, or ill-typed input is provided, the result value will always be \perp . Otherwise, the function translates the CHART operation to the application of a mathematical operation.

We formalize application separately for operations of arity 1, 2, and 3, and then combine these into one application function for arbitrary operations. All application functions are big case distinctions, which explicitly enumerate all the different situations in which the operations can be applied.

Definition 5.2.1: (*apply operation with arity 1*)

The function $apply_1 : \mathcal{O}_1 \times \mathcal{V} \times \mathcal{G}_C \rightarrow \mathcal{V}$ is defined by:

$$apply_1(o, v, G) = \begin{cases} \text{bool } \beta(b = \text{false}) & \text{if } o = \text{not} \wedge v = \text{bool } b \\ \text{bool } \beta(\text{type}_n(v) \leq_{tg(G)} t) & \text{if } o = \text{inst-of } t \wedge v \in \mathcal{N} \\ \text{int } |V| & \text{if } o = \text{size} \wedge v = \text{set } V \\ \text{int } |V| & \text{if } o = \text{size} \wedge v = \text{list } V \\ get_C(v, f, G) & \text{if } o = \text{get-field } f \wedge v \in \mathcal{N} \\ \perp & \text{otherwise} \end{cases}$$

Definition 5.2.2: (apply operation with arity 2)

The function $apply_2 : \mathcal{O}_2 \times \mathcal{V} \times \mathcal{V} \rightarrow \mathcal{V}$ is defined by:

$$\begin{aligned}
 & apply_2(o, v_1, v_2) \\
 = & \left\{ \begin{array}{ll}
 \text{bool } \beta(b = \text{true} \wedge b' = \text{true}) & \text{if } o = \text{and} \wedge v_1 = \text{bool } b \wedge v_2 = \text{bool } b' \\
 \text{int } \lfloor i/j \rfloor & \text{if } o = \text{div} \wedge v_1 = \text{int } i \wedge v_2 = \text{int } j \\
 \text{float } (i/j) & \text{if } o = \text{div} \wedge v_1 = \text{float } i \wedge v_2 = \text{float } j \\
 \text{bool } \beta(v_1 \in V) & \text{if } o = \text{el-of} \wedge v_1 = \text{list } V \\
 \text{bool } \beta(v_1 \in V) & \text{if } o = \text{el-of} \wedge v_1 = \text{set } V \\
 \text{bool } \beta(v_1 = v_2) & \text{if } o = \text{eq} \\
 \text{bool } \beta(i < j) & \text{if } o = \text{lt} \wedge v_1 = \text{int } i \wedge v_2 = \text{int } j \\
 \text{bool } \beta(i < j) & \text{if } o = \text{lt} \wedge v_1 = \text{float } i \wedge v_2 = \text{float } j \\
 \text{int } (i - j) & \text{if } o = \text{minus} \wedge v_1 = \text{int } i \wedge v_2 = \text{int } j \\
 \text{float } (i - j) & \text{if } o = \text{minus} \wedge v_1 = \text{float } i \wedge v_2 = \text{float } j \\
 \text{set } (V \setminus W) & \text{if } o = \text{minus} \wedge v_1 = \text{set } V \wedge v_2 = \text{set } W \\
 \text{int } (i \bmod j) & \text{if } o = \text{mod} \wedge v_1 = \text{int } i \wedge v_2 = \text{int } j \\
 \text{int } (i + j) & \text{if } o = \text{plus} \wedge v_1 = \text{int } i \wedge v_2 = \text{int } j \\
 \text{float } (i + j) & \text{if } o = \text{plus} \wedge v_1 = \text{float } i \wedge v_2 = \text{float } j \\
 \text{set } (V \cup W) & \text{if } o = \text{plus} \wedge v_1 = \text{set } V \wedge v_2 = \text{set } W \\
 \text{set } (\overline{V \cup W}) & \text{if } o = \text{plus} \wedge v_1 = \text{set } V \wedge v_2 = \text{list } W \\
 \text{set } (\overline{V} \cup \overline{W}) & \text{if } o = \text{plus} \wedge v_1 = \text{list } V \wedge v_2 = \text{set } W \\
 \text{list } (V \oplus W) & \text{if } o = \text{plus} \wedge v_1 = \text{list } V \wedge v_2 = \text{list } W \\
 \text{int } (i * j) & \text{if } o = \text{times} \wedge v_1 = \text{int } i \wedge v_2 = \text{int } j \\
 \text{float } (i * j) & \text{if } o = \text{times} \wedge v_1 = \text{float } i \wedge v_2 = \text{float } j \\
 v_{i+1} & \text{if } o = \text{sel-at} \wedge v_1 = \text{list } \langle v'_1 \dots v'_n \rangle \\
 & \wedge v_2 = \text{int } i \wedge 0 \leq i < n \\
 \text{list } \langle v'_1 \dots v'_j \rangle & \text{if } o = \text{sel-upto} \wedge v_1 = \text{list } \langle v'_1 \dots v'_n \rangle \\
 & \wedge v_2 = \text{int } i \wedge j = \min(i + 1, n) \\
 \text{list } \langle v'_j \dots v'_n \rangle & \text{if } o = \text{sel-from} \wedge v_1 = \text{list } \langle v'_1 \dots v'_n \rangle \\
 & \wedge v_2 = \text{int } i \wedge j = \max(1, i + 1) \\
 \perp & \text{otherwise}
 \end{array} \right.
 \end{aligned}$$

Definition 5.2.3: (apply operation with arity 3)

The function $apply_3 : \mathcal{O}_3 \times \mathcal{V} \times \mathcal{V} \times \mathcal{V} \rightarrow \mathcal{V}$ is defined by:

$$\begin{aligned}
 & apply_3(o, v_1, v_2, v_3) \\
 = & \left\{ \begin{array}{ll}
 \text{list } \langle v_{\max(1, i+1)} \dots v_{\min(n, j+1)} \rangle & \text{if } o = \text{between} \\
 & \wedge v_1 = \text{list } \langle v_1 \dots v_n \rangle \\
 & \wedge v_2 = \text{int } i \wedge v_3 = \text{int } j \\
 \perp & \text{otherwise}
 \end{array} \right.
 \end{aligned}$$

Definition 5.2.4: (*apply arbitrary operation*)

The function $apply_{\mathcal{O}} : \mathcal{O} \times \ell(\mathcal{V}) \times \mathcal{G}_C \rightarrow \mathcal{V}$ is defined by:

$$apply_{\mathcal{O}}(o, V, G) = \begin{cases} apply_1(o, v_1, G) & \text{if } o \in \mathcal{O}_1 \wedge V = \langle v_1 \rangle \\ apply_2(o, v_1, v_2) & \text{if } o \in \mathcal{O}_2 \wedge V = \langle v_1, v_2 \rangle \\ apply_3(o, v_1, v_2, v_3) & \text{if } o \in \mathcal{O}_3 \wedge V = \langle v_1, v_2, v_3 \rangle \\ \perp & \text{otherwise} \end{cases}$$

5.3 Evaluate expression

The behavior of an expression (see Definition 4.3.3) is determined by a function that transforms it into a value. Its different alternatives are evaluated as follows:

- A *variable* is looked up into the variable binding.
- A *value* is returned as is.
- A *node set* is computed by filtering reachable nodes based on type.
- An *operation* is applied by first evaluating its arguments, and then applying Definition 5.2.4.
- For a *predicate* application, first its arguments are evaluated. Then the match block of the predicate is invoked¹. If any match was found, the boolean true is returned. Otherwise, the boolean false is returned.

Definition 5.3.1: (*evaluate expression; see 5.3.2 and 5.4.8*)

The evaluation function $eval : \mathcal{X}^{\mathcal{V}} \times \mathcal{E} \times \mathcal{G}_C \rightarrow \mathcal{V}$ is defined by:

$$eval(\Gamma, e, G) = \begin{cases} \Gamma(e) & \text{if } e \in \mathcal{X} \wedge e \in Dom(\Gamma) \\ e & \text{if } e \in \mathcal{V} \\ apply_{\mathcal{O}}(o, eval_{\ell}(\Gamma, E, G), G) & \text{if } e = op \ o \ E \\ apply_{\mathcal{P}}(p, eval_{\ell}(\Gamma, E, G), G) & \text{if } e = pred \ p \ E \\ set \{n \in reach_C(G) \mid n ::_{tg(G)} t\} & \text{if } e = nodeset \ t \\ \perp & \text{otherwise} \end{cases}$$

Definition 5.3.2: (*evaluate list of expressions*)

The function $eval_{\ell} : \mathcal{X}^{\mathcal{V}} \times \ell(\mathcal{E}) \times \mathcal{G}_C \rightarrow \ell(\mathcal{V})$ is defined by:

$$\begin{aligned} eval_{\ell}(\Gamma, \langle \rangle, G) &= \langle \rangle \\ eval_{\ell}(\Gamma, \langle e : E \rangle, G) &= \langle eval(\Gamma, e, G) : eval_{\ell}(\Gamma, E, G) \rangle \end{aligned}$$

5.4 Matching (and predicates)

The behavior of a match block is determined by a function that computes all its possible matches. A single match is a binding of variables to values such that all the equations in the match block are satisfied. Note that for an implementation to continue, it is sufficient to compute one match, or determine that there are no matches at all. The formalization computes *all* matches, however. This is to ensure that a transformation leads to a single deterministic result, regardless of the specific match that was chosen.

¹this may actually be a recursive call, as the application of a predicate may depend on the evaluation of an expression

Below, the matching function will be introduced in a top-down fashion. On the top level, the general structure of the algorithm is as follows:

- The match block is represented as a list of match statements (see Definition 4.4.1), which are processed one by one.
- At each statement, there is both an input set of matches and an output one. The input set represents all the matches that are valid up to that point, and the output set those that are valid afterwards. The algorithm is initialized with a single match, which provides a binding for the fixed rule/predicate arguments.
- A match statement can either be a new variable to look for, a boolean equation, or a lifted forall equation. For match variables, the current set of matches is *extended*. For equations, the current set of matches is *filtered*. For foralls, the current set of matches is also filtered, but a *greatest upper bound* is computed as well.

The top level function that performs this task is formalized as follows:

Definition 5.4.1: (*match*; see 5.4.2, 5.4.3, 5.4.4, 5.4.5 and 5.4.6)

The function $match : \wp(\mathcal{X}^{\mathcal{V}}) \times \mathcal{S}_m \times \mathcal{G}_C \rightarrow \wp(\mathcal{X}^{\mathcal{V}})$ is defined by:

$$\begin{aligned} match(M, \text{search } x, G) &= \cup_{m \in M} [extend(m, x, G)] \\ match(M, \text{check } e, G) &= \cup_{m \in M} [filter(m, x, G)] \\ match(M, \text{forall } x \in S, G) &= gub(e, \cup_{m \in M} [filterAll(m, x, e, S, G)]) \end{aligned}$$

Definition 5.4.2: (*match list*)

The function $match_{\ell} : \wp(\mathcal{X}^{\mathcal{V}}) \times \ell(\mathcal{S}_m) \times \mathcal{G}_C \rightarrow \wp(\mathcal{X}^{\mathcal{V}})$ is defined by:

$$\begin{aligned} match_{\ell}(M, \langle \rangle, G) &= \langle \rangle \\ match_{\ell}(M, \langle s : S \rangle, G) &= match_{\ell}(match(M, s, G), S, G) \end{aligned}$$

When a new match variable is encountered, all the input matches are extended with all the possible values for that variable. The possible extensions of a single match are computed with the *extend* function. The type of the variable is used to determine the valid values.

Definition 5.4.3: (*extend match with all possible variable values*)

The function $extend : \mathcal{X}^{\mathcal{V}} \times \mathcal{X} \times \mathcal{G}_C \rightarrow \wp(\mathcal{X}^{\mathcal{V}})$ is defined by:

$$extend(m, x, G) = \{m[x \mapsto v] \mid v \in \mathcal{V} \mid v ::_{tg(G)} type_x(x) \wedge nodes(v) \subseteq reach_C(G)\}$$

When a new boolean equation is encountered, it is evaluated for all the input matches. If the equation evaluates to true, the match is maintained, and if it evaluates to false, the match is thrown away. This check is performed for a single match by the *filter* function:

Definition 5.4.4: (*filter match by checking equation*)

The function $filter : \mathcal{X}^{\mathcal{V}} \times \mathcal{E} \times \mathcal{G}_C \rightarrow \wp(\mathcal{X}^{\mathcal{V}})$ is defined by:

$$filter(m, e, G) = \begin{cases} \{m\} & \text{if } eval(m, e, G) = \text{bool true} \\ \emptyset & \text{otherwise} \end{cases}$$

When a new forall statement is encountered, two things happen. Assume that the statement is of the form ‘forall (x:E) B’. First, the input matches are filtered. A match m is only kept if the block B has a match for all possible extensions $m[x \mapsto v]$, where $v \in E$. This is accomplished by the *filterAll* function:

Definition 5.4.5: (*filter match by checking forall block*)

The function $filterAll : \mathcal{X}^{\mathcal{V}} \times \mathcal{X} \times \mathcal{E} \times \ell(\mathcal{S}_m) \times \mathcal{G}_C \rightarrow \wp(\mathcal{X}^{\mathcal{V}})$ is defined by:

$$filterAll(m, x, e, S, G) = \begin{cases} \{m\} & \text{if } eval(m, e, G) = \text{list } V \\ & \wedge \forall v \in \mathcal{V} [match_{\ell}(\{m[x \mapsto v]\}, S, G) \neq \emptyset] \\ \{m\} & \text{if } eval(m, e, G) = \text{set } V \\ & \wedge \forall v \in \mathcal{V} [match_{\ell}(\{m[x \mapsto v]\}, S, G) \neq \emptyset] \\ \emptyset & \text{otherwise} \end{cases}$$

The second phase is only carried out if E is a match variable itself (say y). In this case, if $y=S$ is a valid match, then $y=S'$ is also valid match for all $S' \subseteq S$. In our semantics, we are only interested in the biggest S . The function gub , therefore, throws away all matches with a smaller $y=S'$:

Definition 5.4.6: (*greatest upper bound of accumulator in match; see 5.4.7*)

The function $gub : \mathcal{E} \times \wp(\mathcal{X}^{\mathcal{V}}) \rightarrow \wp(\mathcal{X}^{\mathcal{V}})$ is defined by:

$$gub(e, M) = \begin{cases} \{m \mid m \in M \mid \neg \exists m' \in M [m \sqsubseteq_e m']\} & \text{if } e \in \mathcal{X} \\ M & \text{otherwise} \end{cases}$$

Definition 5.4.7: (*smaller relative to accumulator*)

For each $x \in \mathcal{X}$, the relation $\sqsubseteq_x \subseteq \mathcal{X}^{\mathcal{V}} \times \mathcal{X}^{\mathcal{V}}$ is defined by:

$$\begin{aligned} m \sqsubseteq_x m' &\Leftrightarrow Dom(m) = Dom(m') \wedge x \in Dom(m) \\ &\wedge \forall y \in Dom(m) [x \neq y \Rightarrow m(y) = m'(y)] \\ &\wedge |m(x)| < |m'(x)| \end{aligned}$$

A predicate consists of a match block only. When applied, the predicate should return true if one or more matches exist for its match block, and false otherwise. This behavior can now be formalized in terms of the *match* function. The initial input match is the binding of predicate variables to actual argument values.

Definition 5.4.8: (*application of a predicate*)

The function $apply_{\mathcal{P}} : \mathcal{P} \times \ell(\mathcal{V}) \times \mathcal{G}_C \rightarrow \mathcal{V}$ is defined by:

$$apply_{\mathcal{P}}(p, V, G) = \begin{cases} \text{bool true} & \text{if } input_{rs(G)}(p) = \langle x_1 \dots x_n \rangle \\ & \wedge V = \langle v_1 \dots v_n \rangle \\ & \wedge m = \{(x_i, v_i) \mid 1 \leq i \leq n\} \\ & \wedge match_{\ell}(\{m\}, matchb_{rs(G)}(p), G) \neq \emptyset \\ \text{bool false} & \text{otherwise} \end{cases}$$

5.5 Updating

The behavior of an update block is determined by a function that changes a variable binding and a graph according to the changes that are specified in the block. This function can be decomposed into three phases:

1. Sequentially processing the ‘let’ block.
2. Sequentially processing the ‘let’ statements that occur in the ‘set’ block.
3. Simultaneously processing the (evaluated) ‘set’ statements.

These three phases will be described separately in the following subsections.

5.5.1 Processing ‘let’ block

The ‘let’ block consists of a list of update ‘let’ statements (see Definition 4.5.1). Each ‘let’ statement is either the creation of a new node, or the assignment of an expression to a variable. The effect of these actions on a variable binding and a graph can be described directly, as follows:

Definition 5.5.1: (*execution of a ‘let’ statement*)

The function $seq : \mathcal{X}^{\mathcal{V}} \times \mathcal{S}_{u:l} \times \mathcal{G}_C \rightarrow \mathcal{X}^{\mathcal{V}} \times \mathcal{G}_C$ is defined by:

$$seq(\Gamma, \text{assign } x \ e, G) = (\Gamma[x \mapsto \text{eval}(\Gamma, e, G)], G)$$

$$seq(\Gamma, \text{create } x \ t, G) = (\Gamma[x \mapsto n], G')$$

$$\text{where } (n, G') = \text{new}_C(t, G)$$

Definition 5.5.2: (*sequential execution of a list of ‘let’ statements*)

The function $seq_\ell : \mathcal{X}^{\mathcal{V}} \times \ell(\mathcal{S}_{u:l}) \times \mathcal{G}_C \rightarrow \mathcal{X}^{\mathcal{V}} \times \mathcal{G}_C$ is defined by:

$$seq_\ell(\Gamma, \langle \rangle, G) = (\Gamma, G)$$

$$seq_\ell(\Gamma, \langle s : S \rangle, G) = seq_\ell(\Gamma', S, G')$$

$$\text{where } (\Gamma', G') = seq(\Gamma, s, G)$$

5.5.2 Pre-processing ‘set’ block

The ‘set’ block consists of a list of update ‘set’ statements (see Definition 4.5.2). A ‘set’ statement can still contain ‘let’ statements, however, by means of the foreach alternative. These need to be evaluated *before* the other ‘set’ statements are carried out. This is accomplished by the following pre-processing algorithm:

- The algorithm takes a *list* of ‘set’ statements as input, as well as the variable binding and graph that are valid after the ‘let’ block has been processed (i.e. they are the output of seq).
- The output of the algorithm is a set of graph updates, each of the form (node, field, value) or (node, field, index, value). The algorithm does not have a variable binding as output, because the effect of the executed ‘let’ statements is local to each foreach only. For the same reason, the graph is also not part of the output, because the only changes made to it are the creation of local variables.
- If the algorithm encounters a set or seti statement, the expressions in it are evaluated to values, and a graph update is produced as output.
- If the algorithm encounters a foreach, it first evaluates the collection expression to a set of values. For each value, it first processes the ‘let’ statements by means of calling seq , and then continues with a recursive call on the ‘set’ statements.

Definition 5.5.3: (*graph updates*)

The set Upd of graph updates is defined by:

$$Upd = (\mathcal{N} \times \mathcal{T}_f \times \mathcal{V}) \cup (\mathcal{N} \times \mathcal{T}_f \times \mathbb{N} \times \mathcal{V})$$

Definition 5.5.4: (*pre-process ‘set’ statement; see also 5.5.5*)

The function $pre : \mathcal{S}_{u:s} \times (\mathcal{X}^{\mathcal{V}} \times \mathcal{G}_C) \rightarrow \wp(\text{Upd})$ is defined by:

$$pre(\text{set } e_1 \ f \ e_2, \ (\Gamma, G)) = \begin{cases} \{(n, f, v)\} & \text{if } eval(\Gamma, e_1, G) = n \\ & \wedge eval(\Gamma, e_2, G) = v \\ & \wedge n \in \mathcal{N} \\ \emptyset & \text{otherwise} \end{cases}$$

$$pre(\text{set}_i e_1 \ f \ e_2 \ e_3, \ (\Gamma, G)) = \begin{cases} \{(n, f, i, v)\} & \text{if } eval(\Gamma, e_1, G) = n \\ & \wedge eval(\Gamma, e_2, G) = \text{int } i \\ & \wedge eval(\Gamma, e_3, G) = v \\ & \wedge (n \in \mathcal{N}) \wedge (i \in \mathbb{N}) \\ \emptyset & \text{otherwise} \end{cases}$$

$$pre(\text{foreach } x \ e \ S_1 \ S_2, (\Gamma, G)) = \cup_{v \in eval(\Gamma, e, G)} [pre_\ell(S_2, seq(\Gamma, S_1, G))]$$

Definition 5.5.5: (*pre-process list of ‘set’ statements*)

The function $pre_\ell : \ell(\mathcal{S}_{u:s}) \times (\mathcal{X}^{\mathcal{V}} \times \mathcal{G}_C) \rightarrow \wp(\text{Upd})$ is defined by:

$$pre_\ell(S, C) = \cup_{s \in S} [pre(s, C)]$$

5.5.3 Simultaneous application of graph updates

The graph updates that were collected by the pre function (Definition 5.5.4) need to be applied simultaneously. This will be realized by *merging* the set of updates into a single change function of signature $\mathcal{N} \times \mathcal{T}_f \hookrightarrow \mathcal{V}$, which can then be processed in one go by the set function (Definition 3.4.7).

Merging a set of updates is only possible if they are disjoint. This will be checked with the $disj$ predicate, which checks the following conditions:

- There may not be two non-indexed updates of the same field.
- There may not be two indexed updates of the same field at the same index. Two indexed updates with different indexes are allowed, however.
- There may not be both a non-indexed and an indexed update of the same field.

Definition 5.5.6: (*disjointness of sets of graph updates*)

The predicate $disj \subseteq \wp(\text{Upd})$ is defined by:

$$disj(U) \Leftrightarrow \forall_{(n,f,v) \in U} \forall_{(n',f',v') \in U} [n = n' \wedge f = f' \Rightarrow v = v']$$

$$\wedge \forall_{(n,f,i,v) \in U} \forall_{(n',f',i',v') \in U} [n = n' \wedge f = f' \wedge i = i' \Rightarrow v = v']$$

$$\wedge \forall_{(n,f,v) \in U} \forall_{(n',f',i',v') \in U} [n \neq n' \vee f \neq f']$$

Merging a set of updates is formalized by the mrg function. It determines the new value of a field after application of the set of updates, as follows:

- If a non-indexed update of the field exists, the new value is the value that is specified by this update.
- If one or more indexed updates of the field exist, the new value is the old value of the field, but with the elements at the indicated indexes replaced. The indexed replace is carried out by the upi function. Replacing a single element by a list is allowed, and is interpreted as an insert operation.
- If no update of the field exists, the value remains unchanged.

Definition 5.5.7: (*merge disjoint graph updates; see 5.5.8*)

The function $mrg : \wp(\text{Upd}) \times \mathcal{G}_C \rightarrow (\mathcal{N} \times \mathcal{T}_f \leftrightarrow \mathcal{V})$ is defined by:

$$mrg(U, G)(n, f) = \begin{cases} v & \text{if } (n, f, v) \in U \\ \text{list } ftt(\text{upi}(I, 0, V)) & \text{if } I = \{(i, v) \mid (n, f, i, v) \in U\} \\ & \wedge I \neq \emptyset \\ & \wedge \text{get}_C(n, f, G) = \text{list } V \end{cases}$$

Definition 5.5.8: (*process indexed graph updates*)

The function $\text{upi} : \wp(\mathbb{N} \times \mathcal{V}) \times \mathbb{N} \times \ell(\mathcal{V}) \rightarrow \ell(\ell(\mathcal{V}))$ is defined by:

$$\text{upi}(I, i, \langle \rangle) = \langle \rangle$$

$$\text{upi}(I, i, \langle v : V \rangle) = \begin{cases} \langle \langle u \rangle : \text{upi}(I, i+1, V) \rangle & \text{if } (i, u) \in I \\ & \wedge \neg \exists U \in \ell(\mathcal{V}) [u = \text{list } U] \\ \langle U : \text{upi}(I, i+1, V) \rangle & \text{if } (i, \text{list } U) \in I \\ \langle \langle v \rangle : \text{upi}(I, i+1, V) \rangle & \text{otherwise} \end{cases}$$

The merged updates can be applied straightforwardly to the graph. This is formalized by the par function, as follows:

Definition 5.5.9: (*execute a set of merged updates*)

The function $par : \wp(\text{Upd}) \times \mathcal{G}_C \rightarrow \mathcal{G}_C$ is defined by:

$$par(U, G) = \begin{cases} \text{set}_C(mrg(U, G), G) & \text{if } \text{disj}(U) \\ G & \text{otherwise} \end{cases}$$

5.5.4 Executing update block as a whole

The behavior of an update block as a whole can now be described fully. First, seq must be applied to sequentially execute the ‘let’ statements. Then, pre must be applied to also execute the ‘let’ statements in the ‘set’ block, and to transform the rest of the ‘set’ block into a set of graph updates. Finally, par must be applied to simultaneously carry out these graph updates. This combined behavior is formalized as follows:

Definition 5.5.10: (*execute update block*)

The function $\text{upd} : \mathcal{X}^{\mathcal{V}} \times \ell(\mathcal{S}_{u:l}) \times \ell(\mathcal{S}_u) \times \mathcal{G}_C \rightarrow \mathcal{X}^{\mathcal{V}} \times \mathcal{G}$ is defined by:

$$\text{upd}(\Gamma, S_1, S_2, G) = (\Gamma', \text{par}(\text{pre}_\ell(S_2, (\Gamma', G')), G'))$$

where $(\Gamma', G') = \text{seq}(\Gamma, S_1, G)$

Chapter 6

Semantics (sequencing, rule systems)

The behavior of a rule system is determined by computing the set of finite traces through an *automaton*. Each trace represents one execution path of the rule system, which starts at the start rule on the initial graph, and produces a single output graph at the end. The semantics of the rule system is given by the set of possible output graphs, which for a deterministic system should all be equivalent (formally: isomorphic).

In Sections 6.1 and 6.2, the rule system is first transformed into a control automaton, which models abstract execution paths. In Section 6.3, the dynamic behavior of control actions is defined. In Section 6.4, the dynamic behavior is integrated into the control automaton, which results in a system automaton. The semantics of the rule system is defined in terms of the traces through this automaton.

6.1 Control automaton

A control automaton is a special kind of *push-down automaton*, which is *finite* and *deterministic*. The states are represented by tuples of a rule symbol and a list of natural numbers, the transitions are labeled with atomic execution actions, and the stack symbols are control states. It has one initial state, and three distinct final states.

Definition 6.1.1: (*states for a control automaton*)

The set S_c is defined by:

$$S_c = \mathcal{R} \times \ell(\mathbb{N})$$

Definition 6.1.2: (*push and pop transitions*)

The set \mathcal{L}_p is defined by:

$$\begin{aligned} \mathcal{L}_p = & \{\text{push } c \mid c \in S_c\} \\ & \cup \{\text{pop } c \mid c \in S_c\} \end{aligned}$$

Definition 6.1.3: (*control automaton; see Definition 6.1.7*)

An control automaton is a septuple $(S, \Sigma, T, I, U, C, F)$, in which:

- $S \subseteq S_c$ is the set of states of the automaton, which must be finite;
- $\Sigma \subseteq \mathcal{L}_c \cup \mathcal{L}_p$ is the alphabet of the automaton, which must be finite;
- $T : S \times \Sigma \rightarrow S$ is the transition function of the automaton;
- $I \in S$ is the initial state of the automaton; and
- $U, C, F \in S$ are the final states of the automaton.

The universe of control automata is denoted by \mathcal{A}_c .

If $A \in \mathcal{A}_c$, then its components will be denoted with $S_A, \Sigma_A, T_A, A_i, A_U, A_C$ and A_F , respectively. Furthermore, tuples of a state and a stack will be denoted by $S'_A = S_A \times \ell(S_A)$.

The states of the control automaton uniquely represent an execution position. The rule symbol indicates which rule is currently being applied, and the list of natural numbers is an arbitrary representation of the position in the rule. The list allows for easy α -conversion, for instance by adding unique prefixes.

The different final states of the control automaton correspond to the different results of executing a transformation rule: U means ‘graph changed, success’, C means ‘graph changed, success’, and F means ‘graph unchanged, match failed’. The control automaton can continue differently for these three cases, which allows the propagation of rule failure to be defined.

The valid traces of a control automaton are determined by all possible ways to reach a final state (with an empty stack) from the initial state (and an empty stack). The *language* of a control automaton is the set of all its finite traces.

Definition 6.1.4: (*stacked transition function of a control automaton*)

For all $A \in \mathcal{A}_c$, $\rightarrow_A \subseteq S'_A \times \Sigma_A \times S'_A$ is defined by:

$$((c, S), l, (c', S')) \in \rightarrow_A \Leftrightarrow T_A(c, l) = c' \wedge \begin{cases} S' = \langle d : S \rangle & \text{if } l = \text{push } d \\ S = \langle d : S' \rangle & \text{if } l = \text{pop } d \\ S = S' & \text{otherwise} \end{cases}$$

Let $(c, S) \xrightarrow{l}_A (c', S')$ abbreviate $((c, S), l, (c', S')) \in \rightarrow_A$.

Definition 6.1.5: (*traces of a control automaton*)

For all $A \in \mathcal{A}_c$, let $traces_A : \mathbb{N} \times S'_A \times S'_A \rightarrow \wp(\ell(\Sigma_A))$ be defined by:

$$traces_A(n, c, d) = \begin{cases} \{\langle \rangle\} & \text{if } n = 0 \wedge c = d \\ \emptyset & \text{if } n = 0 \wedge c \neq d \\ \{\langle l : L \rangle \mid c \xrightarrow{l}_A c' \wedge \\ L \in traces_A(n-1, c', d)\} & \text{if } n > 0 \end{cases}$$

Definition 6.1.6: (*language of a control automaton*)

The language of a control automaton is defined by:

$$\begin{aligned} L(A) = & \cup_{n \in \mathbb{N}} [traces_A(n, (A_i, \langle \rangle), (A_U, \langle \rangle))] \\ & \cup \cup_{n \in \mathbb{N}} [traces_A(n, (A_i, \langle \rangle), (A_C, \langle \rangle))] \\ & \cup \cup_{n \in \mathbb{N}} [traces_A(n, (A_i, \langle \rangle), (A_F, \langle \rangle))] \end{aligned}$$

The transitions of the control automaton are labeled with atomic execution actions. There are two different kinds of actions:

- Actions for executing a match or update block as a whole. An explicit distinction is made between match success and match failure. This results in three actions: match, nomatch and update.
- Actions for executing sequence statements. A sequence block is not treated as atomic, but is instead simplified into unit actions. These units are:
 - assign, for assignments;
 - cond, for conditions;
 - call and return, for rule calls;
 - pick, for choosing an arbitrary element of a collection variable.

All sequence statements can be expressed in terms of (automata over) these units, see Definitions 6.2.1 and 6.2.2.

In addition, λ is also a valid action, which is used for composing automata.

Definition 6.1.7: (*labels for the control automaton*)

The set \mathcal{L}_c of labels for a control automaton is defined by:

$$\begin{aligned} \mathcal{L}_c = & \{ \text{assign } x \ e \mid x \in \mathcal{X}, e \in \mathcal{E} \} \\ & \cup \{ \text{call } r \ E \mid r \in \mathcal{R}, E \in \ell(\mathcal{E}) \} \\ & \cup \{ \text{return } X \ r \mid X \in \ell(\mathcal{X}), r \in \mathcal{R} \} \\ & \cup \{ \text{cond } e \mid e \in \mathcal{E} \} \\ & \cup \{ \text{pick } x \ y \mid x, y \in \mathcal{X} \} \\ & \cup \{ \text{match } r \mid r \in \mathcal{R} \} \\ & \cup \{ \text{nomatch } r \mid r \in \mathcal{R} \} \\ & \cup \{ \text{update } r \mid r \in \mathcal{R} \} \\ & \cup \{ \lambda \} \end{aligned}$$

For convenience, we will use the following abbreviations:

- ‘cond ! e ’ denotes ‘cond (op not $\langle e \rangle$)’; and
- ‘cond $|e| = 0$ ’ denotes ‘cond (op eq $\langle \text{op size } \langle e \rangle, \text{int } 0 \rangle$)’.

6.2 Building the control automaton

The control automaton for the rule system will be built from the bottom up. First, sequence statements are transformed, then sequence blocks, then individual rules, and finally the rule system as a whole. In this process, smaller automata will frequently be combined. This requires the states of these automata to be disjoint, which is ensured by the following convention:

- Each control automaton is built with an externally provided initial state.
- If the given initial state is (r, L) , then the automaton will only use states of the form $(r, L \oplus L')$. It is the responsibility of the environment to ensure that the prefix (r, L) is unique.
- The final states of the automaton will always be $(r, L \oplus \langle 0 \rangle)$, $(r, L \oplus \langle 1 \rangle)$ and $(r, L \oplus \langle 2 \rangle)$, for U , C and F , respectively. This allows the final states to be referenced from the environment.
- If $I = (r, L)$ is the initial state of the automaton, then I_U , I_C and I_F abbreviate the respective final states (as above). Also, I_n abbreviates $(r, L \oplus \langle n + 3 \rangle)$ for any $n \in \mathbb{N}$.

A sequence statement can contain a rule call. In the control automaton, this is represented by a call transition to the initial state of the rule, and a corresponding return from the final state of the rule. To be able to refer to these states in a rule, the following convention will be used:

- The initial state of a rule r is always $(r, \langle \rangle)$, the U final state is always $(r, \langle 0 \rangle)$, the C final state is always $(r, \langle 1 \rangle)$, and the F final state is always $(r, \langle 2 \rangle)$.
- These states will be denoted by r_i , r_U , r_C and r_F , respectively. Also, let r_n denote $(r, \langle n + 3 \rangle)$.
- Note that these states are the same for each automaton; they are not disjoint, and are merged when the automata are combined.

A single sequence statement can now be transformed into a control automaton. Instead of building the automaton as a whole, only its transition function will be produced, represented as a set. This allows automata to be combined easily, by taking the union of two sets. The initial and final states of the automaton can still be referenced, as the initial state is provided externally, and the final states can be derived from it. The transitions for each sequence statement are as follows:

- An assignment is modeled by a single transition, labeled with itself.

- A rule call to r is modeled by a call transition to r_i , and return transitions from r_U , r_C and r_F . The call transition is parameterized with the rule and its arguments, and the return transition is parameterized with the rule and the caller variables to store the return values. In addition, the call is preceded by a push, and each return is preceded by the matching pop. This distinguishes the returns from different calls. These returns have the same source state, and may also have the same label (i.e. same caller variables).
- An if statement is modeled by two condition transitions (label cond), one for taking the if-branch, and one for taking the else-branch.
- Try and repeat statements do not have transition labels of their own. Instead, they just combine their argument automata in a specific way.
- A foreach statement is modeled by a loop, which first assigns the collection expression to a variable, and then repeatedly extracts a single value out of this variable until it is empty. The extraction is modeled by a specific transition with the label pick.

Definition 6.2.1: (control transitions for a sequence statement; see 6.2.2)

The function $trans : S_c \times \mathcal{S}_s \rightarrow \wp(S_c \times (\mathcal{L}_c \cup \mathcal{L}_p) \times S_c)$ is defined by:

$$\begin{aligned}
trans(I, \text{assign } x \ e) &= \{(I, \text{assign } x \ e, I_U)\} \\
trans(I, \text{apply } X \ r \ E) &= \{(I, \text{push } I, I_1), (I_1, \text{call } r \ E, r_i), (r_F, \text{pop } I, I_F)\} \\
&\cup \{(r_U, \text{pop } I, I_2), (I_2, \text{return } X \ r, I_U), (r_C, \text{pop } I, I_3), (I_3, \text{return } X \ r, I_C)\} \\
trans(I, \text{if } e \ S_1 \ S_2) &= \{(I, \text{cond } e, I_1), (I, \text{cond } !e, I_2), (I_{1C}, \lambda, I_C), (I_{1U}, \lambda, I_U), (I_{1F}, \lambda, I_F) \\
&\quad, (I_{2C}, \lambda, I_C), (I_{2U}, \lambda, I_U), (I_{2F}, \lambda, I_F)\} \\
&\cup trans_\ell(I_1, S_1) \cup trans_\ell(I_2, S_2) \\
trans(I, \text{try } S_1 \ S_2) &= \{(I, \lambda, I_1), (I_{1C}, \lambda, I_C), (I_{1U}, \lambda, I_U), (I_{1F}, \lambda, I_2) \\
&\quad, (I_{2C}, \lambda, I_C), (I_{2U}, \lambda, I_U), (I_{2F}, \lambda, I_F)\} \\
&\cup trans_\ell(I_1, S_1) \cup trans_\ell(I_2, S_2) \\
trans(I, \text{repeat } S) &= \{(I, \lambda, I_1), (I_{1C}, \lambda, I_2), (I_{1U}, \lambda, I_1), (I_{1F}, \lambda, I_U) \\
&\quad, (I_{2C}, \lambda, I_2), (I_{2U}, \lambda, I_2), (I_{2F}, \lambda, I_C)\} \\
&\cup trans_\ell(I_1, S) \cup trans_\ell(I_2, S) \\
trans(I, \text{foreach } x \ e \ S) &= \{(I, \text{assign } y \ e, I_1), (I_1, \text{cond } |y| = 0, I_U), (I_1, \text{pick } x \ y, I_2) \\
&\quad, (I_{2C}, \lambda, I_3), (I_{2U}, \lambda, I_2), (I_{2F}, \lambda, I_F) \\
&\quad, (I_3, \text{cond } |y| = 0, I_C), (I_3, \text{pick } x \ y, I_4) \\
&\quad, (I_{4C}, \lambda, I_4), (I_{4U}, \lambda, I_4), (I_{4F}, \lambda, I_F)\} \\
&\cup trans_\ell(I_2, S) \cup trans_\ell(I_4, S) \\
&\text{where } y \text{ is a new, fresh variable}
\end{aligned}$$

Definition 6.2.2: (control transitions for a list of sequence statements)

The function $trans_\ell : S_c \times \ell(\mathcal{S}_s) \rightarrow \wp(S_c \times (\mathcal{L}_c \cup \mathcal{L}_p) \times S_c)$ is defined by:

$$\begin{aligned}
trans_\ell(I, \langle \rangle) &= \{(I, \lambda, I_U)\} \\
trans_\ell(I, \langle s : S \rangle) &= \{(I, \lambda, I_1), (I_{1C}, \lambda, I_2), (I_{1U}, \lambda, I_3), (I_{1F}, \lambda, I_F) \\
&\quad, (I_{2C}, \lambda, I_C), (I_{2U}, \lambda, I_C) \\
&\quad, (I_{3C}, \lambda, I_C), (I_{3U}, \lambda, I_U), (I_{3F}, \lambda, I_F)\} \\
&\cup trans(I_1, s) \cup trans_\ell(I_2, S) \cup trans_\ell(I_3, S)
\end{aligned}$$

The control transitions for a rule can now be built straightforwardly. The match and update phases are modeled by the special atomic match, nomatch, and update transitions. The sequence phase is modeled by incorporating all the control transitions of its sequence block.

Definition 6.2.3: (*control transitions for a rule*)

The function $trans_{\mathcal{R}} : \mathcal{R} \times \mathcal{RS} \rightarrow \wp(S_c \times (\mathcal{L}_c \cup \mathcal{L}_p) \times S_c)$ is defined by:

$$\begin{aligned}
trans_{\mathcal{R}}(r, R) = & \begin{cases} \{(r_i, \text{match } r, r_1), (r_i, \text{nomatch } r, r_F)\} & \text{if } matchb_R(r) = S \wedge |S| > 0 \\ \{(r_i, \lambda, r_1)\} & \text{otherwise} \end{cases} \\
& \cup \begin{cases} \{(r_1, \text{update } r, r_2)\} & \text{if } updateb_R(r) = (S_1, S_2) \wedge |S_1| + |S_2| > 0 \\ \{(r_1, \lambda, r_2)\} & \text{otherwise} \end{cases} \\
& \cup trans_{\ell}(r_2, sequenceb_R(r)) \\
& \cup \begin{cases} \{(r_{2f}, \lambda, r_C) \mid f \in \{U, C\}\} & \text{if } updateb_R(r) = (S_1, S_2) \wedge |S_1| + |S_2| > 0 \\ \{(r_{2f}, \lambda, r_f) \mid f \in \{U, C, F\}\} & \text{otherwise} \end{cases}
\end{aligned}$$

The control automaton for the rule system as a whole can be determined by taking the union of the control transitions of all its rules, and then turning this into a control automaton. The initial and final states of the automaton are the initial and final states of the start rule.

Definition 6.2.4: (*turn control transitions into control automaton*)

The function $aut : S_c \times S_c \times S_c \times S_c \times \wp(S_c \times (\mathcal{L}_c \cup \mathcal{L}_p) \times S_c) \rightarrow \mathcal{A}_c$ is defined by:

$$\begin{aligned}
aut(I, U, C, F, T) = & (\{s \mid \exists l \in \mathcal{L}_c \exists s' \in S_c [(s, l, s') \in T]\} \cup \{s \mid \exists l \in \mathcal{L}_c \exists s' \in S_c [(s', l, s) \in T]\}, \\
& \{l \mid \exists s, s' \in S_c [(s, l, s') \in T]\}, \\
& T, \\
& I, U, C, F)
\end{aligned}$$

Definition 6.2.5: (*build control automaton of rule system*)

The function $control : \mathcal{RS} \rightarrow \mathcal{A}_c$ is defined by:

$$\begin{aligned}
control(R) = & aut(s_i, s_U, s_C, s_F, \cup_{r \in rules_R} [trans_{\mathcal{R}}(r, R)]) \\
& \text{where } s = start_R
\end{aligned}$$

6.3 Dynamic behavior

The control automaton models *abstract* execution paths only. To make execution concrete, the effect of a trace on an input graph must be computed. This computation needs to maintain a local execution state, which consists of a graph, a current variable binding, and a stack of variable bindings. The stack remembers the variable bindings at the moment of calling a rule, and are needed for evaluating the corresponding return statements.

Definition 6.3.1: (*local execution states*)

The set of local execution states S_e is defined by:

$$S_e = \mathcal{G}_C \times \mathcal{X}^{\mathcal{V}} \times \ell(\mathcal{X}^{\mathcal{V}})$$

An atomic action is not always enabled in a certain execution state (for instance, cond), and can also have a non-deterministic effect (for instance, match and pick). To ensure that each trace represents a unique and valid execution path, we first expand atomic actions into sets of deterministic actions that are known to be enabled in a certain execution state. The deterministic actions are called ‘system actions’, or ‘system labels’, and are represented as follows:

Definition 6.3.2: (*system labels*)

The set \mathcal{L}_s of system labels is defined by:

$$\begin{aligned} \mathcal{L}_s = & (\mathcal{L}_c \mathcal{L}_p) \setminus (\{\text{match } r \mid r \in \mathcal{R}\} \cup \{\text{pick } x \ y \mid x, y \in \mathcal{X}\}) \\ & \cup \{\text{match } r \ \Gamma \mid r \in \mathcal{R}, \Gamma \in \mathcal{X}^\mathcal{V}\} \\ & \cup \{\text{pick } x \ y \ v \ V \mid x, y \in \mathcal{X}, v, V \in \mathcal{V}\} \end{aligned}$$

The function *det* transforms each combination of an execution state and an atomic action into a set of (enabled) system actions. Non-deterministic actions are expanded into all their possibilities, and actions that are not enabled are reduced to the empty set.

Definition 6.3.3: (*determinize actions, i*)

The function $\text{deter} : S_e \times (\mathcal{L}_c \cup \mathcal{L}_p) \mapsto \wp(\mathcal{L}_s)$ is defined by:

$$\begin{aligned} \text{deter}((G, \Gamma, C), \text{cond } e) &= \begin{cases} \{l\} & \text{if } \beta(\text{eval}(\Gamma, e, G)) \\ \emptyset & \text{otherwise} \end{cases} \\ \text{deter}((G, \Gamma, C), \text{pick } x \ y) &= \begin{cases} \{\text{pick } x \ y \ v \ (\text{list } V)\} & \text{if } \Gamma(y) = \text{list } \langle v : V \rangle \\ \{\text{pick } x \ y \ v \ (\text{set } (V \setminus \{v\})) \mid v \in V\} & \text{if } \Gamma(y) = \text{set } V \\ \emptyset & \text{otherwise} \end{cases} \\ \text{deter}((G, \Gamma, C), \text{match } r) &= \begin{cases} \{\text{match } r \ \Gamma' \mid \Gamma' \in \text{match}_\ell(\{\Gamma\}, S, G)\} & \text{if } \text{match}_{rs(G)}(r) = S \\ \emptyset & \text{otherwise} \end{cases} \\ \text{deter}((G, \Gamma, C), \text{nomatch } r) &= \begin{cases} \{l\} & \text{if } \text{match}_{rs(G)}(r) = S \wedge \text{match}_\ell(\{\Gamma\}, S, G) = \emptyset \\ \emptyset & \text{otherwise} \end{cases} \end{aligned}$$

Definition 6.3.4: (*determinize actions, ii*)

The function $\text{det} : S_e \times (\mathcal{L}_c \cup \mathcal{L}_p) \mapsto \wp(\mathcal{L}_s)$ is defined by:

$$\text{det}(S, l) = \begin{cases} \text{deter}(S, l) & \text{if } (S, l) \in \text{Dom}(\text{deter}) \\ \{l\} & \text{otherwise} \end{cases}$$

The dynamic behavior of a system action can now be defined as a function that transforms an execution state into a single new one. This formalized by the *dyn* function, as follows:

Definition 6.3.5: (*dynamic behavior*)

The function $\text{dyn} : S_e \times \mathcal{L}_s \rightarrow S_e$ is defined by:

$$\text{dyn}((G, \Gamma, C), l) = \begin{cases} (G, \Gamma[x \mapsto \text{eval}(\Gamma, e, G)], C) & \text{if } l = \text{assign } x \ e \\ (G, \emptyset[X \mapsto \text{eval}(\Gamma, E, G)], \langle \Gamma : C \rangle) & \text{if } l = \text{call } r \ E \\ & \wedge \text{input}_{rs(G)}(r) = X \\ (G, \Gamma'[X \mapsto \text{eval}(\Gamma, E, G)], C') & \text{if } l = \text{return } X \ r \\ & \wedge C = \langle \Gamma' : C' \rangle \\ & \wedge \text{return}_{rs(G)}(r) = E \\ (G, \Gamma[x \mapsto v][y \mapsto V], C) & \text{if } l = \text{pick } x \ y \ v \ V \\ (G, \Gamma', C) & \text{if } l = \text{match } r \ \Gamma' \\ (G', \Gamma', C) & \text{if } l = \text{update } r \\ & \wedge \text{update}_{rs(G)}(r) = (S, S') \\ & \wedge \text{upd}(\Gamma, S, S', G) = (\Gamma', G') \\ (G, \Gamma, C) & \text{otherwise} \end{cases}$$

6.4 System automaton

A system automaton models dynamic execution paths, and is basically the integration of dynamic behavior into the control automaton. It is a deterministic automaton, with a single initial state and a set of (equivalent) final states. Its states are tuples of control states and execution states, and its transitions are system actions. The system automaton is allowed to have an infinite set of states.

Definition 6.4.1: (*states of the system automaton*)

The set S_s of system states is defined by:

$$S_s = (S_c \times \ell(S_c)) \times S_e$$

Definition 6.4.2: (*get graph out of system state*)

The function $graph : S_s \rightarrow \mathcal{G}_C$ is defined by:

$$graph(S, (G, \Gamma, C)) = G$$

Definition 6.4.3: (*system automaton*)

An system automaton is a quintuple (S, Σ, T, I, F) , in which:

- $S \subseteq S_s$ is the set of states of the automaton;
- $\Sigma \subseteq \mathcal{L}_s$ is the alphabet of the automaton;
- $T : S \times \Sigma \hookrightarrow S$ is the transition function of the automaton;
- $I \in S$ is the initial state of the automaton; and
- $F \subseteq S$ are the final states of the automaton.

The universe of system automata is denoted by \mathcal{A}_s .

If $A \in \mathcal{A}_s$, then its components will be denoted with S_A, Σ_A, T_A, I_A and F_A , respectively.

Furthermore, let $S \xrightarrow{I}_A S'$ abbreviate $T_A(S, I') = S'$.

The function *inc* incorporates dynamic behavior into a single stacked transition of a control automaton. The dynamic behavior is defined for any execution state.

Definition 6.4.4: (*incorporate dynamic behavior in single transition*)

The function $inc : (S_c \times \ell(S_c)) \times (\mathcal{L}_c \cup \mathcal{L}_p) \times (S_c \times \ell(S_c)) \rightarrow \wp(S_s \times \mathcal{L}_s \times S_s)$ is defined by:

$$inc(S_1, l, S_2) = \{((S_1, S_3), l', (S_2, dyn(S_3, l'))) \mid S_3 \in S_e, l' \in det(S_3, l)\}$$

A control automaton can now be enhanced to a system automaton by incorporating behavior in all of its stacked transitions. This process also requires an initial graph, which determines the initial state of the system automaton:

Definition 6.4.5: (*enhance control automaton*)

The function $enhance : \mathcal{G}_C \times \mathcal{A}_c \rightarrow \mathcal{A}_s$ is defined by:

$$enhance(G, A) = (S_s, \mathcal{L}_s, \cup_{t \in \rightarrow_A} [inc(t)] \\ , ((A_i, \langle \rangle), (G, \emptyset, \langle \rangle)) \\ , \{((A_U, \langle \rangle), S) \mid S \in S_s\} \cup \{((A_C, \langle \rangle), S) \mid S \in S_s\} \\)$$

The language of the system automaton is, again, defined as the set of its finite traces between an initial state and one of the final states. For convenience, the intermediate graphs are stored in the traces as well.

Definition 6.4.6: (*traces of a system automaton*)

For all $A \in \mathcal{A}_s$, let $traces_A : \mathbb{N} \times S_A \times S_A \rightarrow \wp(\ell(\Sigma_A \cup \mathcal{G}_C))$ be defined by:

$$traces_A(n, c, d) = \begin{cases} \{\langle graph(c) \rangle\} & \text{if } n = 0 \wedge c = d \\ \emptyset & \text{if } n = 0 \wedge c \neq d \\ \{\langle graph(c), l \rangle \oplus L \mid c \xrightarrow{A} c' \wedge \\ L \in traces_A(n-1, c', d)\} & \text{if } n > 0 \end{cases}$$

Definition 6.4.7: (*language of a system automaton*)

The language of a system automaton is defined by:

$$L(A) = \bigcup_{n \in \mathbb{N}} \bigcup_{f \in F_A} [traces_A(n, I_A, f)]$$

The behavior of a rule system on an initial graph can now be defined. It is given by building the control automaton, then enhancing it with dynamic behavior, then computing all its finite traces, and finally selecting all final graphs from the traces.

Definition 6.4.8: (*meaning of a system automaton*)

The meaning of a system automaton is defined by:

$$\llbracket A \rrbracket = \{G \mid G \in \mathcal{G}_C \mid \exists L \in \ell(\Sigma_A \cup \mathcal{G}_C) [L \oplus \langle G \rangle \in L(A)]\}$$

Definition 6.4.9: (*build system automaton of a rule system*)

The function $system : \mathcal{G}_C \times \mathcal{RS} \rightarrow \mathcal{A}_s$ is defined by:

$$system(G, R) = enhance(G, control(R))$$

Definition 6.4.10: (*apply rule system*)

The function $apply : \mathcal{RS} \times \mathcal{G}_C \rightarrow \wp(\mathcal{G}_C)$ is defined by:

$$apply(R, G) = \llbracket system(G, R) \rrbracket$$