Type checking for a multiset rewriting language

Pascal Fradet and Daniel Le Métayer
IRISA/INRIA
Campus de Beaulieu,
35042 Rennes, France
[fradet,lemetayer]@irisa.fr

Abstract. We enhance Gamma, a multiset rewriting language, with a notion of *structured multiset.* A structured multiset is a set of addresses satisfying specific relations which can be used in the rewriting rules of the program. A type is defined by a context-free graph grammar and a structured multiset belongs to a type T if its underlying set of addresses satisfies the invariant expressed by the grammar defining T. We define a type checking algorithm which allows us to prove mechanically that a program maintains its data structure invariant.

Keywords: multiset rewriting, graph grammars, type checking, invariant, verification.

1 Gamma: motivations and limitations

Gamma is a kernel language which can be introduced intuitively through the chemical reaction metaphor [2, 3]. The unique data structure is the multiset which can be seen as a chemical solution. A simple program is a pair (*Condition*, *Action*) called a reaction. Execution proceeds by replacing in the multiset elements satisfying the condition by the products of the action. The result is obtained when a stable state is reached, that is to say when no more reactions can take place. The following is an example of a Gamma program computing the maximum element of a non-empty set.

$$max \ : \ x \ , \ y \ , \ x \leq y \ \Rightarrow \ y$$

$x \leq y$ specifies a property to be satisfied by the selected elements x and y. These elements are replaced in the set by the value y. Nothing is said in this definition about the order of evaluation of the comparisons. If several disjoint pairs of elements satisfy the condition, the reactions can be performed in parallel. Let us consider as another introductory example, a sorting program. We represent a sequence as a set of pairs $(index, value)$ and the program exchanges ill-ordered values until a stable state is reached and all values are well-ordered.

$$sort \ : \ (i, x) \ , \ (j, y) \ , \ (i < j) \ , \ (x > y) \ \Rightarrow \ (i, y) \ , \ (j, x)$$

The interested reader may find in [2] a longer series of examples (string processing problems, graph problems, geometry problems, ...) illustrating the Gamma style of programming and in [3] a review of contributions related to the chemical reaction model. The fundamental quality of the language is the possibility of writing programs without any artificial sequentiality (sequentiality which is not imposed by the logic of the program). This makes program derivation easier [1] and naturally leads to the construction of programs suitable for execution on parallel machines. But our experience with Gamma has also highlighted some weaknesses of the language. In particular, the language does not make it easy to structure data or to specify particular control strategies. An unfortunate consequence is that the programmer sometimes has to resort to tricky encodings to express his algorithm. For instance, the exchange sort algorithm shown above is expressed in terms of multisets of pairs (*index,value*). Also, it is difficult to reach a decent level of efficiency in any general purpose implementation of Gamma, partly because the underlying structure of the data (and control) is not exposed to the compiler. Such information could be exploited to improve the implementation [5] but it can usually not be recovered by an automatic analysis of the program. So, the lack of structuring facility is detrimental both for reasoning about programs and for implementing them. In this paper, we propose a solution to this problem which does not jeopardize the basic qualities of the language. Our proposal is based on a notion of *structured multiset* which is a set of addresses satisfying specific relations and associated with values. The relations express a form of neighborhood between the molecules of the solution; they can be used in the reaction condition of a program or transformed by the action. In our framework, a type is defined in terms of rewrite rules on the relations of a multiset. A structured multiset belongs to a type T if its underlying set of addresses satisfies the invariant expressed by the rewrite system defining T. The paper defines a type checking algorithm which allows us to prove mechanically that a program maintains its data structure invariant.

We define the notion of structured multiset and structured program in section 2. The notion of structuring types is introduced in section 3 with a collection of examples illustrating the programming style of Structured Gamma. In section 4, we describe a checking algorithm and illustrate it with several examples. The conclusion presents several applications built upon the ideas developed in this paper.

2 Syntax and semantics of Structured Gamma

A structured multiset is a set of addresses satisfying specific relations. As an example, the list [5; 2; 7] can be represented by a structured multiset whose set of addresses is $\{a_1,\ a_2,\ a_3\}$ and associated values are $Val(a_1) = 5,\ Val(a_2) = 2,\ Val(a_3) = 7$. Let **succ** be a binary relation and **end** a unary relation; the addresses satisfy

$$\textbf{succ}\ a_1\ a_2\ ,\ \textbf{succ}\ a_2\ a_3\ ,\ \textbf{end}\ a_3$$

A Structured Gamma program is defined in terms of pairs of a condition and an action which can:

- test/modify the relations on addresses,
- test/modify the values associated with addresses.

As an illustration, an exchange sort for lists can be written in Structured Gamma as:

$$\mathit{Sort} = \mathbf{succ}\ x\ y\ ,\ \overline{x} > \overline{y}\ \ \Rightarrow\ \ \mathbf{succ}\ x\ y\ ,\ x\ :=\ \overline{y}\ ,\ y\ :=\ \overline{x}$$

The two selected addresses x and y must satisfy the relation $\mathbf{succ}\ x\ y$ and their values $\overline{x}$ and $\overline{y}$ are such that $\overline{x} > \overline{y}$. The action exchanges their values and leaves the relation unchanged.

In order to define the syntax and semantics of Structured Gamma, we consider three basic domains:

- $\mathbf{R}$: set of relation symbols,
- $\mathbf{A}$: set of addresses,
- $\mathbf{V}$: set of values.

We note $\mathbf{A}(M)$ the set of addresses occurring in the multiset M.

Syntax The syntax of Structured Gamma programs is described by the following grammar:

$$\begin{array}{ll} <\mathit{Program}> & ::= \text{ProgName} = [<\mathit{Reaction}>]^* \\ <\mathit{Reaction}> & ::= <\mathit{Condition}>\ \Rightarrow\ <\mathit{Action}> \\ <\mathit{Condition}> & ::= \mathbf{r}\ x_1 \ldots x_n \mid f^{\text{Bool}}(\overline{x_1}, \ldots, \overline{x_n})\ \mid <\mathit{Condition}>, <\mathit{Condition}> \\ <\mathit{Action}> & ::= \mathbf{r}\ x_1 \ldots x_n \mid x := f^{\mathbf{V}}(\overline{x_1}, \ldots, \overline{x_n})\ \mid\ <\mathit{Action}>, <\mathit{Action}> \end{array}$$

where $\mathbf{r}$ ($\in \mathbf{R}$) denotes a n-ary relation, x_i is an address variable, $\overline{x_i}$ is the value at address x_i and $f^{\mathbf{X}}$ is a function from $\mathbf{V}^n$ to $\mathbf{X}$.

As can be seen in the *Sort* example, $\overline{x}$ refers to the value of address x when it is selected in the multiset. The evaluation order of the basic operations of an action (in particular, assignments) is not semantically relevant. In order to fit with this design choice, a valid Structured Gamma program must satisfy two additional syntactic conditions:

- If $\overline{x}$ occurs in the reaction then x occurs in the condition.
- An action may not include two assignments to the same variable.

Semantics A structured multiset M can be seen as $M = \mathit{Rel} + \mathit{Val}$ where

- Rel is a *multiset* of relations represented as tuples $(\mathbf{r}, a_1, \ldots, a_n)$ $(\mathbf{r} \in \mathbf{R}, a_i \in \mathbf{A})$
- Val is a *set* of values represented by triplets of the form $(\mathbf{val}, a, v)$ $(a \in \mathbf{A}, v \in \mathbf{V})$

For example, the structured multiset shown at the beginning of this section can be noted:

$$\{(\mathbf{succ}, a_1, a_2), (\mathbf{succ}, a_2, a_3), (\mathbf{end}, a_3), (\mathbf{val}, a_1, 5), (\mathbf{val}, a_2, 2), (\mathbf{val}, a_3, 7)\}$$

A valid structured multiset is such that an address x does not have more than one value (i.e. x occurs at most once in Val). On the other hand, there may be several occurrences of the same tuple in Rel. Also, we do not enforce that

$$\mathbf{A}(Rel) \subseteq \mathbf{A}(Val) \text{ nor that } \mathbf{A}(Val) \subseteq \mathbf{A}(Rel)$$

So, allocated addresses may not to possess a value or may have a value but not occur in any relation (although, in this case, they cannot be accessed by a Structured Gamma program and may be garbage collected).

In order to define the semantics of programs, we associate three functions with each reaction $C \Rightarrow A$:

– a boolean function $\mathcal{T}(C)$ representing the condition of application of a reaction:
$$\mathcal{T}(C)(a_1, \ldots, a_i, b_1, \ldots, b_j) = (\mathbf{val}, a_1, \overline{a_1}) \in Val \ \wedge \ldots \wedge \ (\mathbf{val}, a_i, \overline{a_i}) \in Val$$
$$\wedge \ (\mathbf{val}, b_1, \overline{b_1}) \in Val \ \wedge \ldots \wedge \ (\mathbf{val}, b_j, \overline{b_j}) \in Val \ \wedge \lfloor C \rfloor$$
– A function $\mathcal{C}(C)$ representing the tuples selected by the condition (i.e. the relations and values occurring in C):
$$\mathcal{C}(C)(a_1, \ldots, a_i, b_1, \ldots, b_j) = \{(\mathbf{val}, a_1, \overline{a_1}), \ldots, (\mathbf{val}, a_i, \overline{a_i}),$$
$$(\mathbf{val}, b_1, \overline{b_1}), \ldots, (\mathbf{val}, b_j, \overline{b_j})\} + \lceil C \rceil$$
– A function $\mathcal{A}(A)$ representing the tuples added by the action (i.e. the relations occurring in A, the values selected but unchanged by the reaction and assigned values):
$$\mathcal{A}(A)(a_1, \ldots, a_i, b_1, \ldots, b_j, c_1, \ldots, c_k) = \{(\mathbf{val}, a_1, \overline{a_1}), \ldots, (\mathbf{val}, a_i, \overline{a_i})\} + \lceil A \rceil$$

where

– $(a_1, \ldots, a_i)$ denotes the set of non-assigned variables whose value occurs in the reaction,
– $(b_1, \ldots, b_j)$ denotes the set of assigned variables occurring in the condition C,
– $(c_1, \ldots, c_k)$ denotes the set of variables occurring only in the action A.

and $\lceil \ \rceil$ and $\lfloor \ \rfloor$ are defined by:

$$\begin{array}{ll} \lceil X_1, X_2 \rceil = \lceil X_1 \rceil + \lceil X_2 \rceil & \lfloor X_1, X_2 \rfloor = \lfloor X_1 \rfloor \wedge \lfloor X_2 \rfloor \\ \lceil \mathbf{r}\ x_1 \ldots x_n \rceil = \{(\mathbf{r},\ x_1, \ldots, x_n)\} & \lfloor \mathbf{r}\ x_1 \ldots x_n \rfloor = (\mathbf{r},\ x_1, \ldots, x_n) \in Rel \\ \lceil f(\overline{x_1}, \ldots, \overline{x_n}) \rceil = \emptyset & \lfloor f(\overline{x_1}, \ldots, \overline{x_n}) \rfloor = f(\overline{x_1}, \ldots, \overline{x_n}) \\ \lceil x := f(\overline{x_1}, \ldots, \overline{x_n}) \rceil = \{(\mathbf{val}, x, f(\overline{x_1}, \ldots, \overline{x_n}))\} & \end{array}$$

The semantics of a Structured Gamma program $P = C_1 \Rightarrow A_1, \ldots, C_m \Rightarrow A_m$ applied to a multiset M is defined as the set of normal forms of the following rewrite system:

$$M \longrightarrow_P \mathcal{GC}(M) \quad \text{if } \forall\{x_1, \ldots, x_n\} \subseteq \mathbf{A}(M) \quad \forall i \in [1 \ldots m] \quad \neg\mathcal{T}(C_i)(x_1, \ldots, x_n)$$

$$M \longrightarrow_P M - \mathcal{C}(C_i)(x_1, \ldots, x_n) + \mathcal{A}(A_i)(x_1, \ldots, x_n, y_1, \ldots, y_k) \text{ (with } y_i \notin \mathbf{A}(M))$$
$$\text{if } \exists\{x_1, \ldots, x_n\} \subseteq \mathbf{A}(M) \text{ and } \exists i \in [1 \ldots m] \text{ such that } \mathcal{T}(C_i)(x_1, \ldots, x_n)$$

If no tuple of addresses satisfies any condition then a normal form is found. The result is the accessible structure described by the relations. Addresses which do not occur in Rel are removed from Val. More precisely:

$$\mathcal{GC}(Rel + Val) = Rel + \{(\mathbf{val}, a, v) \mid (\mathbf{val}, a, v) \in Val \ \wedge \ a \in \mathbf{A}(Rel)\}$$

Otherwise, a tuple of addresses $(x_1, \ldots, x_n)$ and a pair (C_i, A_i) such that $\mathcal{T}(C_i)(x_1, \ldots, x_n)$ are non-deterministically chosen. The multiset is transformed by removing $\mathcal{C}(C_i)(x_1, \ldots, x_n)$, allocating fresh addresses $y_1, \ldots, y_k$ and adding $\mathcal{A}(A_i)(x_1, \ldots, x_n, y_1, \ldots, y_k)$.

Correspondence between Structured Gamma and original Gamma Compared to the original Gamma formalism, the basic model of computation remains unchanged. It still consists in repeated applications of local actions in a global data structure. Actually, our way to define the semantics of Structured Gamma programs is very close to a translation into equivalent pure Gamma programs.

Rather than providing a formal definition of the translation, we illustrate it with the exchange sort program which is defined as follows in Structured Gamma:

$$Sort = \mathbf{succ}\ x\ y\ ,\ \overline{x} > \overline{y} \quad \Rightarrow \mathbf{succ}\ x\ y\ ,\ x\ := \ \overline{y}\ ,\ y\ := \ \overline{x}$$

and can be rewritten in pure Gamma as:

$$\begin{aligned} Sort = {} & (\mathbf{val}, x, \overline{x})\ ,\ (\mathbf{val}, y, \overline{y})\ ,\ (\mathbf{succ}, x, y)\ ,\ \overline{x} > \overline{y} \\ & \Rightarrow\ (\mathbf{succ}, x, y)\ ,\ (\mathbf{val}, x, \overline{y})\ ,\ (\mathbf{val}, y, \overline{x}) \end{aligned}$$

3 Structuring types

Structured multisets can be seen as a syntactic facility allowing the programmer to make the organization of the data explicit. We are now in a position to introduce a new notion of type which characterizes the structure of a multiset. We provide a formal definition of types and we illustrate them with a collection of examples.

3.1 Syntax and semantics of structuring types

Syntax The syntax of types is defined by the following grammar:

$$\begin{array}{ll} <\textit{TypeDeclaration}> & ::= \text{TypeName} = <\textit{Prod}>\ ,\ [<\textit{NonTerminal}> = <\textit{Prod}>]^* \\ <\textit{NonTerminal}> & ::= \text{NTName}\ x_1 \ldots x_n \\ <\textit{Prod}> & ::= \mathbf{r}\ x_1 \ldots x_n \mid\ <\textit{NonTerminal}> \mid <\textit{Prod}>\ ,\ <\textit{Prod}> \end{array}$$

where $\mathbf{r}$ ($\in \mathbf{R}$) is an n-ary relation ($n > 0$), and x_i is a variable denoting an address.

A type definition resembles a context-free graph grammar. For example, lists can be defined as

$$\begin{array}{ll} \textit{List} & = L\ x \\ L\ x & = \mathbf{succ}\ x\ y\ ,\ L\ y \\ L\ x & = \mathbf{end}\ x \end{array}$$

The definition of a type T can be associated with a rewrite system (noted $\leadsto_T$) which can fold any multiset of type T in the start symbol T. It amounts to reverse the grammar rules and to consider '$=$' symbols as '$\leadsto_T$' and nonterminal names N as relations. We keep the same notation $NTx_1 \ldots x_p$ to denote a nonterminal in a type definition or a relation in the rewrite system associated with a type. The correct interpretation is usually clear from the context. For example, the rewrite system associated with the type *List* is noted:

$$\begin{array}{ll} L\ x & \leadsto_{List}\ \textit{List} \\ \mathbf{succ}\ x\ y\ ,\ L\ y & \leadsto_{List}\ L\ x \\ \mathbf{end}\ x & \leadsto_{List}\ L\ x \end{array}$$

This system rewrites any multiset of type List into the atom *List*.

We note $\mid M \mid$ the multiset restricted to relations ($\mid Rel + Val \mid = Rel$).

Definition 1 *A multiset M has type T (noted M:T) iff* $\mid M \mid \ \overset{*}{\leadsto}_T \{T\}$.

Let us point out that $\leadsto_T$ reductions must enforce that if a variable of the *lhs* does not occur in the *rhs* it does not occurs in the rest of the multiset. This is a global operation and such rewriting systems are clearly not Structured Gamma programs. This global condition is induced by the semantics of Structured Gamma programs which enforces variables of the *rhs* not occurring in the *lhs* to be fresh.

3.2 Examples of types

Abstract types found in functional languages such as ML can be defined in a natural way in Structured Gamma. For example, the type corresponding to binary trees is

$$\begin{array}{ll} \textit{Bintree} & = B\ x \\ B\ x & = \mathbf{node}\ x\ y\ z\ ,\ B\ y\ ,\ B\ z \\ B\ x & = \mathbf{leaf}\ x \end{array}$$

However, structuring types make it possible to define not only tree shaped but also graph structures. Actually, the main blessing of the framework is to allow concise definitions of complicated pointer-like structures. To give a few examples, it quite easy to define common imperative structures such as

– doubly-linked lists:

$$\begin{aligned} &Doubly = L\ x \\ &L\ x \quad\ \ = \mathbf{succ}\ x\ y\ ,\ \mathbf{pred}\ y\ x\ ,\ L\ y \\ &L\ x \quad\ \ = \mathbf{end}\ x \end{aligned}$$

– lists with connections to the last element:

$$\begin{aligned} &Last \ \ = L\ x\ z \\ &L\ x\ z = \mathbf{succ}\ x\ y\ ,\ \mathbf{last}\ x\ z\ ,\ L\ y\ z \\ &L\ x\ z = \mathbf{succ}\ x\ z\ ,\ \mathbf{last}\ x\ z\ ,\ \mathbf{end}\ z \end{aligned}$$

– circular lists:

$$\begin{aligned} &Circular = L\ x \\ &L\ x \qquad\ = L'\ x\ z\ ,\ L'\ z\ x \\ &L\ x \qquad\ = \mathbf{succ}\ x\ x \\ &L'\ x\ y \quad\ = L'\ x\ z\ ,\ L'\ z\ y \\ &L'\ x\ y \quad\ = \mathbf{succ}\ x\ y \end{aligned}$$

– binary trees with linked leaves:

$$\begin{aligned} &Binlinked = L\ x\ y\ z \\ &L\ x\ y\ z \quad\ = \mathbf{left}\ x\ u\ ,\ L\ u\ y\ v\ ,\ R\ x\ v\ z \\ &L\ x\ y\ z \quad\ = \mathbf{left}\ x\ y\ ,\ R\ x\ y\ z \\ &R\ x\ y\ z \quad\ = \mathbf{right}\ x\ u\ ,\ \mathbf{succ}\ y\ v\ ,\ L\ u\ v\ z \\ &R\ x\ y\ z \quad\ = \mathbf{right}\ x\ z\ ,\ \mathbf{succ}\ y\ z \end{aligned}$$

3.3 Programming using structuring types

Many programs are expressed more naturally in Structured Gamma than in pure Gamma. The underlying structure of the multiset can be described by a type whereas in pure Gamma we had to encode it using tuples and tags. Let us give a few examples of Structured Gamma programs whose description in pure Gamma is cumbersome. Note that the syntax of programs is extended to account for typed programs (ProgName : TypeName = ...).

Iota takes a singleton $[a]$ and yields the list $[\overline{a}; \overline{a-1}; \ldots; 1]$.

$$Iota\ :\ List = \mathbf{end}\ a\ ,\ \overline{a} > 1 \quad \Rightarrow \quad \mathbf{succ}\ a\ b\ ,\ \mathbf{end}\ b\ ,\ b\ :=\ \overline{a} - 1$$

MultB takes a binary tree and yields a leaf whose value is the product of all the nodes and leaves values of the original tree.

$$MultB\ :\ Bintree = \mathbf{node}\ a\ b\ c\ ,\ \mathbf{leaf}\ b\ ,\ \mathbf{leaf}\ c \quad \Rightarrow \quad \mathbf{leaf}\ a\ ,\ a\ :=\ \overline{a} * \overline{b} * \overline{c}$$

In order to get more potential parallelism, we may also add the rules

$$\begin{array}{lcl}
\mathbf{node}\ a\ b\ c\ ,\ \mathbf{node}\ b\ d\ e,\ \mathbf{leaf}\ c & \Rightarrow & \mathbf{node}\ a\ d\ e,\ a\ :=\ \overline{a} * \overline{b} * \overline{c} \\
\mathbf{node}\ a\ b\ c\ ,\ \mathbf{leaf}\ b\ ,\ \mathbf{node}\ c\ d\ e & \Rightarrow & \mathbf{node}\ a\ d\ e,\ a\ :=\ \overline{a} * \overline{b} * \overline{c}
\end{array}$$

Types can also be used to express precise control constraints. For example, lists can be defined with two identified elements used as pointers to enforce a specific reduction strategy.

$$\begin{array}{ll}
List_m & = L_0\ x \\
L_0\ x & = \mathbf{m1}\ x\ ,\ \mathbf{succ}\ x\ y\ ,\ L_1\ y \\
L_0\ x & = \mathbf{succ}\ x\ y\ ,\ L_0\ y \\
L_1\ x & = \mathbf{m2}\ x\ ,\ L_2\ x \\
L_1\ x & = \mathbf{succ}\ x\ y\ ,\ L_1\ y \\
L_2\ x & = \mathbf{succ}\ x\ y\ ,\ L_2\ y \\
L_2\ x & = \mathbf{end}\ x
\end{array}$$

The type definition enforces that **m1** identifies a list element located before the list element marked by **m2**. Assuming an initial list where **m1** marks the first element and **m2** the second one, we can describe a sequential sort.

$$\begin{array}{ll}
\multicolumn{2}{l}{SeqSort\ :\ List_m =} \\
\mathbf{m1}\ a\ ,\ \mathbf{m2}\ b\ ,\ \overline{a} > \overline{b} & \Rightarrow \mathbf{m1}\ a\ ,\ \mathbf{m2}\ b\ ,\ a\ :=\ \overline{b}\ ,\ b\ :=\ \overline{a} \\
\mathbf{m1}\ a\ ,\ \mathbf{m2}\ b\ ,\ \mathbf{succ}\ b\ c\ ,\ \overline{a} \leq \overline{b} & \Rightarrow \mathbf{m1}\ a\ ,\ \mathbf{m2}\ c\ ,\ \mathbf{succ}\ b\ c \\
\mathbf{m1}\ a\ ,\ \mathbf{m2}\ b\ ,\ \mathbf{end}\ b\ , & \\
\mathbf{succ}\ a\ c\ ,\ \mathbf{succ}\ c\ d\ ,\ \overline{a} \leq \overline{b} & \Rightarrow \mathbf{m1}\ c\ ,\ \mathbf{m2}\ d\ ,\ \mathbf{end}\ b\ ,\ \mathbf{succ}\ a\ c\ ,\ \mathbf{succ}\ c\ d
\end{array}$$

In fact, $List_m$ can be shown more precisely to be a refinement of *List*. We come back to this issue in the conclusion.

To summarize, Structured Gamma retains the spirit of Gamma while providing means to declare data structures and to enforce specific reduction strategies (e.g. for efficiency purposes).

4 Static type checking

The natural question following the introduction of a new type system concerns the design of an associated type checking algorithm. In the context of Structured Gamma, type checking must ensure that a program maintains the underlying structure defined by a type. It amounts to the proof of an invariant property. We propose a checking algorithm based on the construction of an abstract reduction graph which summarizes all possible reduction chains from a condition C to a unique nonterminal. We describe its application to some examples, suggesting that the algorithm is precise enough to tackle most common cases.

4.1 The basic idea

First, let us note that values and assignments are not relevant for type checking. So, we may consider multisets and rewriting rules restricted to relations. Also, we assume that checking is done relatively to a given type T.

A reduction step of a multiset by a Structured Gamma program is of the form $M + C \longrightarrow_P M + A$ where C and A represent multisets of relations matching a reaction of the program. The algorithm has to check that the application of every reaction of the program leaves the type of the multiset unchanged. In other terms, for any reaction $C \Rightarrow A$ and multiset $M + C$ of type T it checks that $M + A$ is of type T (i.e. $M + A \overset{*}{\leadsto}_T \{T\}$).

The checking algorithm is based on the observation that if $M + C$ has type T, there must be a context X ($X \subseteq M$) such that $C + X$ reduces by $\leadsto_T$ to a unique nonterminal $NT\ x_1 \ldots x_p$ (possibly T). The reduction of a term $C (= C_0)$ to a nonterminal $NT\ x_1\ \ldots\ x_p$ can be described as

$$C_0 + X_0 \leadsto_T C_1 \quad C_1 + X_1 \leadsto_T C_2 \quad \ldots \quad C_n + X_n \leadsto_T \{NT\ x_1\ \ldots\ x_p\}$$

Each step is an application of a $\leadsto_T$ rule which reduces at least a component of C_i and X_i is a *basic context.* Basic contexts are the smallest (possibly empty) multisets of relations needed to match the *lhs* of a reduction rule. They are therefore completely reduced by the reduction rule.

The context $X = X_0 + \ldots + X_n$ must be produced by the reduction of M, that is

$$M \overset{*}{\leadsto}_T M' + X$$

and the $\leadsto_T$ reduction of the multiset $M + C$ can then be described as

$$M + C \overset{*}{\leadsto}_T M' + X + C \overset{*}{\leadsto}_T M' + \{NTx_1 \ldots x_p\} \overset{*}{\leadsto}_T \{T\}$$

Now, if $A + X$ reduces to the same unique nonterminal $NTx_1 \ldots x_p$, then

$$M + A \overset{*}{\leadsto}_T M' + X + A \overset{*}{\leadsto}_T M' + \{NTx_1 \ldots x_p\} \overset{*}{\leadsto}_T \{T\}^1$$

and the type of the multiset is maintained.

It is sufficient to check the property $A + X \overset{*}{\leadsto}_T \{NTx_1 \ldots x_p\}$ for every possible reduction chain from C to a nonterminal $NTx_1 \ldots x_p$ with context X. To get round the problem posed by the unbounded length of such chains, we consider residuals C_i up to renaming of variables.

4.2 A checking algorithm

The type checking algorithm consists in examining in turn each reaction of the program.

$$\mathsf{TypeCheck}\ (P, T) = \forall (C, A) \text{ of } P.\ \mathsf{Check}\ (A, T, \mathsf{Build}\ (C, \{C\}, T))$$

For each reaction $C \Rightarrow A$, a reduction graph summarizing all possible reduction chains from C to a nonterminal is built by Build. Then, Check verifies that for any reduction chain and context X of the graph from C to a nonterminal,

[1] The global conditions on this reduction are ensured by the validity of the reduction of $M + C$ and the fact that variables of A are either variables of C or fresh variables.

$A+X$ reduces to the same nonterminal. These functions are described in Figure 1.

Build takes an initial graph made of the root C. The reduction graph is such that nodes are residuals C_i which are all different (even up to renaming of variables) and edges are of the form $C_i \xrightarrow{X,\sigma} C_j$. This notation indicates that $C_i + X \rightsquigarrow_T \sigma C_j$ where X is a basic context and σ is a variable renaming.

```
Build $(C, G, T)$
if $C$ is a nonterminal then return $G$ else
let $CX = \{(C_i, X_i) \mid C + X_i \rightsquigarrow_T C_i\}$ in     $CX$ is a finite set
                                                          (up to fresh variable renaming)
for each $(C_i, X_i)$ in $CX$ do
        if $\exists C_j \in G$ and $\sigma_j$ such that $C_i = \sigma_j C_j$ then $G := G + C \xrightarrow{X_i,\sigma_j} C_j$
        else $G := G + C_i + C \xrightarrow{X_i,id} C_i$ ; $G :=$ Build$(C_i, G, T)$
od
return $G$

Check $(A, T, G)$
let $\mathcal{S} = \{(X_0 + \sigma_1 X_1 + \ldots + \sigma_1 \circ \ldots \circ \sigma_n\ X_n, \sigma_1 \circ \ldots \circ \sigma_{n+1}\ \{NT\ x_1 \ldots x_p\})$
              $\mid C_0 \xrightarrow{X_0,\sigma_1} C_1 \xrightarrow{X_1,\sigma_2} \ldots C_n \xrightarrow{X_n,\sigma_{n+1}} \{NT\ x_1 \ldots x_p\} \in G\}$
and $\mathcal{C} = \{(X_0 + \sigma_1 X_1 + \ldots + \sigma_1 \circ \ldots \circ \sigma_{i-1}\ X_{i-1}, \sigma_1 \circ \ldots \circ \sigma_i\ C_i)$
              $\mid C_0 \xrightarrow{X_0,\sigma_1} \ldots \xrightarrow{X_{i-1},\sigma_i} C_i \in G$ and $\exists C_i \xrightarrow{X,\sigma_x} \ldots C_i \in G$
                 and $\exists C_i \xrightarrow{Y,\sigma_y} \ldots N \in G$ ($N$ a nonterminal)
                 and $\nexists\ j < i \mid C_j \xrightarrow{Z,\sigma_z} \ldots C_j \in G\}$
in $\forall (X, Y) \in \mathcal{S} \cup \mathcal{C}$. Reduces_to $(A + X, Y, T)$

Reduces_to (X, Y, T)
if X=Y then True
else if X is irreducible then False
else let {$X_1$, ..., $X_n$} the set of all possible residuals of X by a $\rightsquigarrow_T$ reduction
      in   $\bigvee_{i=1}^{n}$ Reduces_to ($X_i$, Y, T)
```

Fig. 1. Type checking functions

The structure of Build is a depth first traversal of all possible reduction chains. The recursion stops when C is a nonterminal or is already present in the graph. CX is the set of basic contexts and residuals denoting all the different $\rightsquigarrow_T$ reductions of C. If a residual C_i in already present in the graph, that is, there is already a node C_j such that $C_i = \sigma_j C_j$, then the edge $C \xrightarrow{X_i,\sigma_j} C_j$ is added to the graph. Otherwise, a new node C_i is created and the edge $C \xrightarrow{X_i,id} C_i$ is added.

The function Check takes the graph as argument and performs the following verifications:

- For every simple path from the root to a nonterminal N with context X, it checks that $A + X \overset{*}{\rightsquigarrow}_T N$.

  Let us focus on the meaning of a path $C_0 \xrightarrow{X_0,\sigma_1} C_1 \ldots \xrightarrow{X_n,\sigma_{n+1}} \{NT\ x_1 \ldots x_p\}$. By definition, we have

  $$C_0 + X_0 \rightsquigarrow_T \sigma_1 C_1, \ldots, C_n + X_n \rightsquigarrow_T \sigma_{n+1}\{NT\ x_1 \ldots x_p\}$$

  thus

  $$C_0 + X_0 + \sigma_1 X_1 + \ldots + \sigma_1 \circ \ldots \circ \sigma_n\ X_n \overset{*}{\rightsquigarrow}_T \sigma_1 \circ \ldots \circ \sigma_{n+1}\{NT\ x_1 \ldots x_p\}$$

  So, the context and nonterminal (X, N) associated with the above path are $X = X_0 + \sigma_1 X_1 + \ldots + \sigma_1 \circ \ldots \circ \sigma_n\ X_n$ and $N = \sigma_1 \circ \ldots \circ \sigma_n \circ \sigma_{n+1}\ \{NT\ x_1 \ldots x_p\}$.
- For every simple path with context X from the root to a residual C_i belonging to a cycle, it checks that $A + X \overset{*}{\rightsquigarrow}_T C_i$. In fact, it is sufficient to check this property for the first residual belonging to a cycle occurring on the path from the root and only for cycles which may lead to a nonterminal.

The verifications that the action A with context X can be reduced to Y are implemented by function Reduces_to$(A + X, Y, T)$. It simply tries all the $\rightsquigarrow_T$ reductions on the term $A + X$ using a depth first strategy. If a path leading to Y is found then True is returned. If Reduces_to finds out that all the normal forms of $A + X$ by "$\rightsquigarrow_T$" are different from Y, it returns False which entails the failure of the verification (TypeCheck(P, T) = False).

The termination of TypeCheck is ensured by the following observations:

- The reduction graph is finite. Using the context-free nature of types and equality up to renaming, it is easy to show that the number of nodes and edges is bounded.
- It is possible to find a well-founded decreasing ordering for $\rightsquigarrow_T$ reductions ensuring the termination of Reduces_to.

The type checking is correct if it ensures that the type of a program is invariant throughout the reduction. The proof amounts to showing a subject reduction property.

Property 1 $\forall P,\ M_1 : T\ \ M_1 \longrightarrow_P M_2$ *and* TypeCheck$(P, T) \Rightarrow M_2 : T$

The proof of this property and a more detailed presentation of the algorithm can be found in a companion paper [6].

4.3 Examples

Even if the theoretical complexity of the algorithm is prohibitive, the cost seems reasonable in practice. We take here a few examples to illustrate the type checking process at work.

Example 1. Let us take the *Iota* program working on type *List*. The program is

$$Iota\ :\ List = \textbf{end}\ a\ ,\ \overline{a} > 1 \quad \Rightarrow \quad \textbf{succ}\ a\ b\ ,\ \textbf{end}\ b\ ,\ b\ :=\ \overline{a} - 1$$

Operations on values are not relevant for type checking and we consider the single reduction rule

$$\textbf{end}\ a \quad \Rightarrow \quad \textbf{succ}\ a\ b\ ,\ \textbf{end}\ b$$

The type definition and associated rewriting system are:

$$\begin{array}{ll} List = L\ x & \\ L\ x & = \textbf{succ}\ x\ y\ ,\ L\ y \\ L\ x & = \textbf{end}\ x \end{array} \qquad \begin{array}{ll} L\ x & \rightsquigarrow_{List} List \\ \textbf{succ}\ x\ y\ ,\ L\ y & \rightsquigarrow_{List} L\ x \\ \textbf{end}\ x & \rightsquigarrow_{List} L\ x \end{array}$$

The type checking amounts to the call

$$\mathsf{Check}((\textbf{succ}\ a\ b,\ \textbf{end}\ b),\ List,\ \mathsf{Build}(\textbf{end}\ a,\ \{\textbf{end}\ a\},\ List))$$

There is a single $\rightsquigarrow_{List}$ reduction of **end** $a+X$ (with X a basic context), namely

$$\textbf{end}\ a \rightsquigarrow_{List} L\ a$$

So, $CX = \{(L\ a, \emptyset)\}$ and, since $L\ a$ is a nonterminal, the reduction graph is

$$\textbf{end}\ a \xrightarrow{\emptyset, id} L\ a$$

There is a single simple path and we are left with checking
Reduces_to $((\textbf{succ}\ a\ b\ ,\ \textbf{end}\ b), L\ a, List)$
The set of possible residuals of $(\textbf{succ}\ a\ b\ ,\ \textbf{end}\ b)$ is $\{(\textbf{succ}\ a\ b\ ,\ L\ b)\}$, so Reduces_to $((\textbf{succ}\ a\ b\ ,\ L\ b),\ L\ a\ ,\ List)$ is recursively called. The set of possible residuals of $(\textbf{succ}\ a\ b\ ,\ L\ b)$ by a $\rightsquigarrow_{List}$ reduction is $\{L\ a\}$ and Reduces_to $(L\ a, L\ a, List)$ = True. So, TypeCheck$(Iota, List)$ = True and we conclude that the "List" invariant is maintained.

Example 2. Let us consider a program performing an insertion at the end of a list of type *Last*.
Wrong : *Last* =
$\textbf{succ}\ x\ z\ ,\ \textbf{last}\ x\ z\ ,\ \textbf{end}\ z\ \Rightarrow \textbf{succ}\ x\ z\ ,\ \textbf{succ}\ z\ t\ ,\ \textbf{last}\ x\ t\ ,\ \textbf{last}\ z\ t\ ,\ \textbf{end}\ t$

Obviously this program is ill-typed. If the list has more than two elements, the first elements would still point to z whereas t is the new last element. The definition of the rewriting system of *Last* is:

$$\begin{array}{ll} L\ x\ z & \rightsquigarrow_{Last}\ Last \\ \textbf{succ}\ x\ y\ ,\ \textbf{last}\ x\ z\ ,\ L\ y\ z & \rightsquigarrow_{Last}\ L\ x\ z \\ \textbf{succ}\ x\ z\ ,\ \textbf{last}\ x\ z\ ,\ \textbf{end}\ z & \rightsquigarrow_{Last} L\ x\ z \end{array}$$

There is a single reduction sequence from the condition to a nonterminal:

$$\textbf{succ}\ x\ z\ ,\ \textbf{last}\ x\ z\ ,\ \textbf{end}\ z\ \leadsto_{Last} L\ x\ z$$

but,

$$\textbf{succ}\ x\ z\ ,\ \textbf{succ}\ z\ t\ ,\ \textbf{last}\ x\ t\ ,\ \textbf{last}\ z\ t\ ,\ \textbf{end}\ t \not\leadsto_{Last} L\ x\ z$$

and the “Last” invariant is not maintained.

However, if we consider the insertion program:

$$\mathit{Add : Last}\ = \textbf{succ}\ x\ y,\ \textbf{last}\ x\ z\ \Rightarrow \textbf{succ}\ x\ t,\ \textbf{succ}\ t\ y,\ \textbf{last}\ x\ z,\ \textbf{last}\ t\ z$$

There is one reduction sequence from the condition to a nonterminal:

$$\textbf{succ}\ x\ y\ ,\ \textbf{last}\ x\ z\ ,\ L\ y\ z\ \leadsto_{Last} L\ x\ z$$

and it is easy to check that

$$\begin{aligned}\textbf{succ}\ x\ t,\ \textbf{succ}\ t\ y,\ \textbf{last}\ x\ z,\ \textbf{last}\ t\ z,\ L\ y\ z &\leadsto_{Last} \textbf{succ}\ x\ t,\ \textbf{last}\ x\ z,\ L\ t\ z\\ &\leadsto_{Last}\ L\ x\ z\end{aligned}$$

and the “Last” invariant is maintained.

5 Conclusion

We have applied the framework developed in this paper in three different areas: program refinement, coordination and type systems for imperative languages. We sketch these applications in turn.

- Structured Gamma can serve as a basis for program refinements leading to efficient implementations. The basic source of inefficiency of any “naïve” implementation of Gamma is the combinatorial explosion entailed by the semantics of the language for the selection of reacting elements. As pointed out in [5], most of the refinements leading to efficient optimizations of Gamma programs can be expressed as specific selection orderings. Several refinements are introduced in [5] which shows that they often lead to efficient well-known implementations of the corresponding algorithms. This result is quite satisfactory from a formal point of view because it shows that there is a continuum from specifications written in Gamma to lower-level and efficient program descriptions. These refinements, however, had to be checked manually. Using Structured Gamma as a basis, we can provide general conditions ensuring the correctness of program refinements [6]. The basic idea, which was already alluded to in section 3.3, consists in considering multiset (and type) refinements as the addition of extra relations between addresses. These relations are used as further constraints on the control in order to impose a specific ordering for the selection of elements.
- Coordination languages [4, 9], software architecture languages [8] and configuration languages [12] were proposed as a way to make large applications more manageable and more amenable to formal verifications. They are based

on the principle that the definition of a software application should make a clear distinction between individual components and their interaction in the overall software organization. We have used Structured Gamma as a coordination language by interpreting the addresses in the multisets as names of individual entities to be coordinated [13]. Their associated value defines their behavior (in a given programming language which is independent of the coordination language) and the relations correspond to communication links. A structuring type provides a description of the shape of the overall architecture. An important advantage of our approach is that coordinators can be checked statically (using the algorithm of section 4) to ensure that they preserve the style of the architecture.

- The type systems currently available for imperative languages are too weak to detect a significant class of programming errors. For example, they cannot express the property that a list is doubly-linked or circular. Such structures can be specified naturally using a subset of structuring types that we call "shape types". We have proposed a syntax for a smooth integration of shape types in C [7]. Shapes are manipulated by reactions which can be statically checked and translated in pure C. The programmer can still express pointer manipulations with the expected constant time execution and benefit from the additional guarantee that the property specified by the structuring type is an invariant of the program. The graph types approach [11] shares the same concern. In their framework, a graph is defined using a canonical spanning tree (called the backbone) and auxiliary pointers. Only the backbone can be manipulated by programs and some simple operations may implicitly involve non-constant updates of the auxiliary pointers. In contrast, shape types do not privilege any part of the graph and all operations on the structure appear explicitly in the rewrite rules.

Directions for further research include other application areas such as the specification of networks of processors and various extensions like the use of context-sensitive grammars to describe structuring types.

Acknowledgments. This work was supported by Esprit Basic Research project 9102 *Coordination.*

References

1. J.-P. Banâtre and D. Le Métayer, *The Gamma model and its discipline of programming*, Science of Computer Programming, Vol. 15, pp. 55-77, 1990.
2. J.-P. Banâtre and D. Le Métayer. Programming by multiset transformation, *Communications of the ACM*, Vol. 36-1, pp. 98-111, January 1993.
3. J.-P. Banâtre and D. Le Métayer. Gamma and the chemical reaction model: ten years after, *Coordination programming: mechanisms, models and semantics*, Imperial College Press, 1996.
4. N. Carriero and D. Gelernter, Linda in context, *Communications of the ACM*, Vol. 32-4, pp. 444-458, April 1989.

5. C. Creveuil. *Techniques d'analyse et de mise en œuvre des programmes Gamma*, Thesis, University of Rennes I, 1991.
6. P. Fradet and D. Le Métayer, Structured Gamma, *Irisa Research Report PI-989*, March 1996.
7. P. Fradet and D. Le Métayer, Shape types, In *Proc. of the 24rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1997.
8. D. Garlan and D. Perry, Editor's Introduction, *IEEE Transactions on Software Engineering, Special Issue on Software Architectures*, 1995.
9. A. A. Holzbacher, A software environment for concurrent coordinated programming, *Proc. First int. Conf. on Coordination Models, Languages and Applications*, Springer Verlag, LNCS 1061, pp. 249-266, April 1996.
10. P. Inverardi and A. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model, *IEEE Transactions on Software Engineering*, Vol. 21, No. 4, pp. 373-386, April 1995.
11. N. Klarlund and M. Schwartzbach. Graph types. In *Proc. 20th Symp. on Princ. of Prog. Lang.*, pp. 196-205. ACM, 1993.
12. J. Kramer, Configuration programming. A framework for the development of distributable systems, *Proc. COMPEURO'90*, IEEE, pp. 374-384, 1990.
13. D. Le Métayer, Software architecture styles as graph grammars, in *Proc. of the ACM SIGSOFT'96 4th Symposium on the Foundations of Software Engineering*, 1996, pp. 15-23.