PUBLICATION INTERNE
N^{o} 989

STRUCTURED GAMMA

PASCAL FRADET AND DANIEL LE MÉTAYER

ISSN 1166-8687

Institut de Recherche en Informatique et Systèmes Aléatoires
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 99 84 71 00 – Fax : (33) 99 84 71 71

Structured Gamma

Pascal Fradet and Daniel Le Métayer

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lande

Publication interne n ° 989 — mars 1996 — 17 pages

Abstract: The Gamma language is based on the chemical reaction metaphor which has a number of benefits with respect to parallelism and program derivation. But the original definition of Gamma does not provide any facility for data structuring or for specifying particular control strategies. We address this issue by introducing a notion of *structured multiset* which is a set of addresses satisfying specific relations and associated with a value. The relations can be seen as a form of neighbourhood between the molecules of the solution; they can be used in the reaction condition of a program or transformed by the action. 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 grammar defining T. We define a type checking algorithm which allows us to prove mechanically that a program maintains its data structure invariant. We illustrate the significance of the approach for program reasoning and program refinement.

Key-words: multiset rewriting, type checking, invariant, verification, refinement

(Résumé : tsvp)

*[fradet,lemetayer]@irisa.fr

Centre National de la Recherche Scientifique
(URA 227) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Gamma Structuré

Résumé : Le langage Gamma repose sur la métaphore de la réaction chimique, ce qui lui donne un certain nombre d'avantages en terme de parallélisme et de dérivation de programmes. Cependant, la définition originelle de Gamma n'offre pas de moyen de décrire des structures de données ou de spécifier des stratégies de contrôle particulières. Nous proposons une solution à ce problème en introduisant une notion de *muti-ensemble structuré*. Il s'agit d'un ensemble d'adresses satisfaisant des relations données. Les relations décrivent une forme de voisinage entre les molécules; elles peuvent être utilisées dans une condition de réaction ou transformées par une action. Un type est défini par une grammaire de graphes non-contextuelle. Un multi-ensemble structuré appartient à un type T si l'ensemble d'adresses sous-jacent satisfait l'invariant exprimé par la grammaire définissant le type T. Nous définissons un algorithme de vérification de types qui permet de montrer automatiquement qu'un programme maintient l'invariant de sa structure de données. Nous montrons l'intérêt de cette démarche pour la vérification et le raffinement de programmes.

Mots-clé : récriture de multi-ensemble, vérification de types, invariant, vérification, raffinement

1 Gamma: motivations and limitations

The fast evolution of hardware and the growing needs of end-users has placed new requirements on the design of programming languages: sequentiality should no longer be seen as the prime programming paradigm but just as one of the possible forms of cooperation between individual entities. The Gamma formalism was proposed ten years ago precisely to capture the intuition of computation as the global evolution of a collection of atomic values interacting freely. Gamma is a kernel language which can be introduced intuitively through the chemical reaction metaphor. The unique data structure in Gamma 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 possibility of getting rid of artificial sequentiality in Gamma has two important consequences:

- It confers a very high level nature to the language and allows the programmer to describe programs in a very abstract way.
- Because Gamma programs do not have any sequential bias, the language naturally leads to the construction of parallel programs (in fact, it is much harder to write a sequential program than a parallel program in Gamma).

However, our experience with Gamma also highlighted some weaknesses of the language. Let us now review the most important ones.

- The original definition of Gamma lacks of any operation for combining programs.
- The language does not make it easy for the programmer to structure data or to specify particular control strategies.
- Because of the combinatorial explosion imposed by its semantics, it is difficult to reach a decent level of efficiency in any general purpose implementation of the language.

For the sake of modularity it is desirable that a language offers a rich set of operators for combining programs. It is also fundamental that these operators enjoy a useful collection of algebraic laws in order to make it possible to reason about programs. This issue was addressed in [7, 8] which introduce operators for the parallel and the sequential composition of programs and study their properties and in [11, 5] which define higher-order extensions of Gamma. Another approach was taken in [4] where a notion of *schedules* is proposed to control the execution of Gamma programs.

The lack of support for structuring data and the difficulty of imposing a particular control strategy should not be surprising since the original motivation for the language was to be able to describe programs exhibiting as few ordering constraints as possible. An unfortunate consequence however 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)*. This limitation also introduces an unnecessary factor of inefficiency in the implementation because the underlying structure of the data (and control) is not exposed to the compiler. Such information could be exploited to improve the implementation [6] 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 without jeopardising the basic qualities of the language. Let us point out in particular that it would not be acceptable to take the usual view of recursive type definitions because this would lead to a recursive style of programming and ruin the fundamental locality principle (because the data structure would then be manipulated as a whole). Our proposal is based on a notion of *structured multiset* which is a set of addresses satisfying specific relations and associated with a value. The relations express a form of neighbourhood 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 illustrate the significance of the approach for program reasoning and program refinement.

We define the notion of structured multiset and structured program in section 2. We describe the syntax and a formal semantics of this extension of Gamma and suggest how Structured Gamma programs can be translated in a straightforward way into original Gamma programs. 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 show its correctness. The correctness property is akin to the subject reduction property of type systems for functional languages. We illustrate the type system and type checking algorithm with several examples. Section 5 introduces notions of type and program refinement which can be used to derive efficient implementations from Gamma specifications. Section 6 reviews related proposals in different contexts and suggests several extensions.

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

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

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 :

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

The two selected addresses x and y must satisfy the relation **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:

- **R**: set of relation symbols,
- **A**: set of addresses,
- **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 :

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

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 $M = Rel + Val$ 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 $\lfloor\ \ \rfloor$ and $\lceil\ \ \rceil$ are defined by :

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

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:

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

$$M \Leftrightarrow\!\!\rightarrow_P M \Leftrightarrow \mathcal{C}(C_i)(x_1, \ldots, x_n) + \mathcal{A}(A_i)(x_1, \ldots, x_n, y_1, \ldots, y_k) \quad (\text{with } y_1, \ldots, y_k \notin \mathbf{A}(M))$$

$$\text{if } \exists\{x_1, \ldots, x_n\} \subseteq \mathbf{A}(M) \text{ and } \exists i \in [1 \ldots m] \text{ such that } \mathcal{T}(C_i)(x_1, \ldots, x_n)$$

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:

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

We use the notation $M \overset{*}{\Leftrightarrow\!\!\rightarrow}_P M'$ for $M \overset{*}{\Leftrightarrow\!\!\rightarrow}_P M'$ and M' is a normal form for P.

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 $\mathcal{A}(A_i)(x_1, \ldots, x_n, y_1, \ldots, y_k)$.

Note that the semantics enforces that different variable names denote different addresses. Sometimes, this requirement may lead to unnecessary verbose programs. For example, if we want to express the rewriting of any instance of a relation tuple $(\mathbf{r}, x_1, \ldots, x_n)$, we would like to write $\mathbf{r}\ x_1 \ldots x_n \Rightarrow \ldots$ assuming x_i and x_j may possibly denote the same address rather than enumerating all the possible sharing patterns. Let us note, however, that it is always possible to translate the rule above into an equivalent set of rules where variables cannot be identified. So, a sensible option would be address the matter at the syntax level and add a special notation to denote that some variables may be identified. For example, $\mathbf{r}\ x\_1\ y\_2\ z\_1\ t\_2$ would mean that x and z may be equal, y and t may be equal but x and z are different from y and t. This syntax could be automatically translated into standard rules.

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:

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

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 define a type in terms of rewrite rules on the relations of the multiset. A structured multiset is said to belong to a type if its underlying set of addresses can be produced by the rewrite system defining the type. 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:

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

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

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

The definition of a type T can be associated with a Structured Gamma program (noted $BigBang^T$) which can return any multiset of type T. It amounts to considering '=' symbols as '$\Rightarrow$' and nonterminal names N as relations. We keep the same notation $NTx_1 \dots x_p$ 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 Structured Gamma program associated with the type *List* is noted :

$$BigBang^{List} = \begin{array}{lcl} List & \Rightarrow & L\ \ x \\ L\ \ x & \Rightarrow & \mathbf{succ}\ x\ y\ ,\ L\ \ y \\ L\ \ x & \Rightarrow & \mathbf{end}\ x \end{array}$$

This program applied to a multiset containing only the atom *List* can produce all the finite lists .

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* $\{T\} \overset{*}{\Leftrightarrow\!\!\!\rightarrow}_{BigBang^T} \mid M \mid$.

Let us now introduce the inverse of *BigBang*, called *BigCrunch*, which provides a useful alternative definition of types.

Definition 2 $M \Leftrightarrow\!\!\!\rightarrow_{BigBang^T} M' \Leftrightarrow \quad M' \Leftrightarrow\!\!\!\rightarrow_{BigCrunch^T} M$

Property 1 *A multiset M has type T iff* $\mid M \mid \overset{*}{\Leftrightarrow\!\!\!\rightarrow}_{BigCrunch^T} \{T\}$.

Let us point out that *BigCrunch* 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 *BigCrunch* rewriting systems are clearly not Structured Gamma programs. This global condition is induced by the semantics of Structured Gamma programs (hence *BigBang* 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

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

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 :

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

- lists with connections to the last element :

$$\begin{array}{lcl}
\mathit{Listlast} & = & L\ x\ z \\
L\ x\ z & = & \mathbf{succ}\ x\ y\ ,\ \mathbf{last}\ x\ z\ ,\ L\ y\ z \\
L\ x\ z & = & \mathbf{succ}\ x\ z\ ,\ \mathbf{last}\ x\ z\ ,\ \mathbf{end}\ z
\end{array}$$

- binary trees with linked leaves :

$$\begin{array}{lcl}
\mathit{Binlinked} & = & B\ x\ x'\ x' \\
B\ x\ x'\ x'' & = & \mathbf{node}\ x\ y\ z\ ,\ B\ y\ x'\ y'\ ,\ B\ z\ y'\ x'' \\
B\ x\ x\ x' & = & \mathbf{leaf}\ x\ ,\ \mathbf{succ}\ x\ x'
\end{array}$$

Let us point out that we have assumed that different variables denote different addresses in type definitions (as we did for program definitions). This choice entails the same drawbacks and calls for the same solution as in the case of programs. For example, using the notation hinted at in section 2, circular lists can be defined by:

$$\begin{array}{lcl}
\mathit{Circular} & = & L\ x\ x \\
L\ x\_1\ y\_1 & = & L\ x\ z\ ,\ L\ z\ y \\
L\ x\_1\ y\_1 & = & \mathbf{succ}\ x\ y
\end{array}$$

which is expended into the following type in the pure Structured Gamma syntax:

$$\begin{array}{lcl}
\mathit{Circular} & = & L\ x\ x \\
L\ x\ x & = & L\ x\ z\ ,\ L\ z\ x \\
L\ x\ x & = & \mathbf{succ}\ x\ x \\
L\ x\ y & = & L\ x\ z\ ,\ L\ z\ y \\
L\ x\ y & = & \mathbf{succ}\ x\ y
\end{array}$$

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 $[\overline{a}; \overline{a \Leftrightarrow 1}; \ldots; 1]$.

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

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.

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

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.

$$\begin{array}{lcl}
\mathit{List}_m & = & L_0\ x \\
L_0\ x & = & \mathbf{m1}\ x\ ,\ \mathbf{succ}\ x\ y\ ,\ L_1\ y \\
L_0\ x & = & \mathbf{succ}\ x\ y\ ,\ L_0\ y \\
L_1\ x & = & \mathbf{m2}\ x\ ,\ L_2\ x \\
L_1\ x & = & \mathbf{succ}\ x\ y\ ,\ L_1\ y \\
L_2\ x & = & \mathbf{succ}\ x\ y\ ,\ L_2\ y \\
L_2\ x & = & \mathbf{end}\ x
\end{array}$$

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.

$$
\begin{array}{l}
\textit{SeqSort} \; : \; \textit{List}_m = \\
\begin{array}{lcl}
\mathbf{m1}\; a\;,\; \mathbf{m2}\; b\;,\; \overline{a} > \overline{b} & \Rightarrow & \mathbf{m1}\; a\;,\; \mathbf{m2}\; b\;,\; a := \overline{b}\;,\; b := \overline{a} \\
\mathbf{m1}\; a\;,\; \mathbf{m2}\; b\;,\; \mathbf{succ}\; b\; c\;,\; \overline{a} \leq \overline{b} & \Rightarrow & \mathbf{m1}\; a\;,\; \mathbf{m2}\; c\;,\; \mathbf{succ}\; b\; c \\
\mathbf{m1}\; a\;,\; \mathbf{m2}\; b\;,\; \mathbf{end}\; b\;,\; \mathbf{succ}\; a\; c\;,\; \mathbf{succ}\; c\; d\;,\; \overline{a} \leq \overline{b} & \Rightarrow & \mathbf{m1}\; c\;,\; \mathbf{m2}\; d\;,\; \mathbf{end}\; b\;,\; \mathbf{succ}\; a\; c\;,\; \mathbf{succ}\; c\; d
\end{array}
\end{array}
$$

In fact, $List_m$ can be shown more precisely to be a refinement of *List*. We come back to this issue in section 5.

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 prove the correctness of the algorithm and we describe its application to some examples.

4.1 A checking algorithm

First, let us note that values and assignments are not relevant for type checking. So, in this section, we consider multisets and rewriting rules restricted to relations. Also, we assume a given type T and we use the notation $\rightsquigarrow$ for $\Leftrightarrow\!\!\!\rightarrow_{BigCrunch^T}$.

A reduction step of a multiset by a Structured Gamma program is of the form $M + C \Leftrightarrow\!\!\!\rightarrow_P M + A$ 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. $M + A \overset{*}{\rightsquigarrow} \{T\}$).

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 $BigCrunch^T$ 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

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

Each step is an application of a $BigCrunch^T$ 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

$$M \overset{*}{\rightsquigarrow} M' + X$$

and the *BigCrunch* reduction of the multiset $M + C$ can then be described as

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

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

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

and the type of the multiset is maintained.

[1]The global conditions on this *BigCrunch* 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.

It is sufficient to check the property $A + X \overset{*}{\rightsquigarrow} \{NTx_1 \dots x_p\}$ for every possible reduction chain from C to a nonterminal $NTx_1 \dots x_p$ 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.

A renaming is a one-to-one mapping and its domain is the set of variables which differ from their image. We will use the following lemma

Lemma 1 *Let σ a renaming then $C_1 \rightsquigarrow C_2 \Leftrightarrow \sigma C_1 \rightsquigarrow \sigma C_2$*

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, $A + X$ reduces to the same nonterminal.

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 $C_i \overset{X,\sigma}{\Leftrightarrow\!\!\!\rightarrow} C_j$. This notation indicates that $C_i + X \rightsquigarrow \sigma C_j$ where X is a basic context and σ is a variable renaming. Recall that *BigCrunch* reductions have a global condition: variables suppressed by a reduction rule should not occur in the rest of the multiset. To generate valid *BigCrunch* reduction chains we enforce that variables occurring in a basic context are either variables occurring in the current residual or fresh variables. This way, we never reintroduce suppressed variables.

```
Build $(C,G,T)$
if $C$ is a nonterminal then return $G$ else
let $CX = \{(C_i,X_i) \mid C + X_i \rightsquigarrow C_i\}$ 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 $G := G + C \overset{X_i,\sigma_j}{\Leftrightarrow\!\!\!\rightarrow} C_j$
        else $G := G + C_i + C \overset{X_i,id}{\Leftrightarrow\!\!\!\rightarrow} C_i$ ; $G :=$ Build$(C_i,G,T)$
od
return $G$
```

The structure of the algorithm 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 *BigCrunch*^{T} reductions of C. Note that basic contexts X_i and residuals C_i may occur several times in CX (there may be several possible reduction rules for the same term and different terms can be reduced in the same residual). However, the set CX is finite since pairs (C_i,X_i) are considered up to renaming of fresh variables introduced by X_i.

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 $C \overset{X_i,\sigma_j}{\Leftrightarrow\!\!\!\rightarrow} C_j$ is added to the graph. Otherwise, a new node C_i is created and the edge $C \overset{X_i,id}{\Leftrightarrow\!\!\!\rightarrow} C_i$ 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 $A + X \overset{*}{\rightsquigarrow} N$.

  Let us focus on the meaning of a path $C_0 \overset{X_0,\sigma_1}{\Leftrightarrow\!\!\!\rightarrow} C_1 \dots \overset{X_n,\sigma_{n+1}}{\Leftrightarrow\!\!\!\rightarrow} \{NT\ x_1 \dots x_p\}$.

  By definition, we have $C_0 + X_0 \rightsquigarrow \sigma_1 C_1, \dots, C_n + X_n \rightsquigarrow \sigma_{n+1}\ \{NT\ x_1 \dots x_p\}$ and by lemma 1 we have

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

  So, the context and nonterminal (X,N) associated with the above path are $X = X_0 + \sigma_1 X_1 + \dots + \sigma_1 \circ \dots \circ \sigma_n\ X_n$ and $N = \sigma_1 \circ \dots \circ \sigma_n \circ \sigma_{n+1}\ \{NT\ x_1 \dots x_p\}$.

- For every simple path with context X from the root to a residual C_i belonging to a cycle, it checks that $A + X \overset{*}{\rightsquigarrow} C_i$. 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.

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

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 *BigCrunch*^{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 "$\rightsquigarrow$" are different from Y, it returns False which entails the failure of the verification (TypeCheck(P, T) = False).

```
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 $BigCrunch^T$ reduction
    in    $\bigvee_{i=1}^{n}$ Reduces_to (X$_i$, Y, T)
```

The termination of TypeCheck is ensured by the following observations :

- The reduction graph is finite.
  - The number of nodes is bounded. Since *rhs* of *BigCrunch*^{T} rules are always a single element (non-terminal) the number of relations in C_i's never grows. The number of relation names in a type and in a condition C as well as the arity of relations are bounded so the number of different C_i (up to renaming of variables different from $Var(C)$) is bounded.
  - The number of edges is bounded. For any term C_i there is only a finite number of basic contexts matching a *BigCrunch* rule (up to renaming of fresh variables), and for each basic context there is a finite number of different *BigCrunch* reductions.
- Reduces_to terminates. It is always possible to find a well-founded decreasing ordering for *BigCrunch* reductions. As usual with context-free grammars, it is always possible to put the type definition in Chomsky normal form and all *BigCrunch* rules would be of the form

  $$NT_1 \; x_1 \ldots x_i \; , \; NT_2 \; y_1 \ldots y_j \rightsquigarrow NT_3 \; z_1 \ldots z_k \quad \text{or} \quad \mathbf{r} \; x_1 \ldots x_i \rightsquigarrow NT \; y_1 \ldots y_j$$

  Let $nt(T)$ and $nnt(T)$ denote the number of terminals and nonterminals of T respectively, then $T_1 \ll T_2$ iff $nt(T_1) < nt(T_2)$ or $(nt(T_1) = nt(T_2)$ and $nnt(T_1) < nnt(T_2))$ is a well-founded ordering.

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 2 $\forall P, \; M_1 : T \quad M_1 \Leftrightarrow\!\!\!\!\rightarrow_P M_2$ *and* TypeCheck$(P, T) \Rightarrow M_2 : T$

Proof. A rewriting $M_1 \Leftrightarrow\!\!\!\!\rightarrow_P M_2$ involves a rule $C \Rightarrow A$ and can be described as $M_1 = M + C \Leftrightarrow\!\!\!\!\rightarrow M + A = M_2$. Since $M + C$ is a multiset of type T then, by definition, there is a reduction

$$M + C \overset{*}{\rightsquigarrow} M' + X_0 + \ldots + X_n + C \overset{*}{\rightsquigarrow} M' + NTx_1 \ldots x_p \overset{*}{\rightsquigarrow} \{T\}$$

$$\text{with} \qquad C(= C_0) + X_0 \rightsquigarrow C_1 \quad C_1 + X_1 \rightsquigarrow C_2 \quad \ldots \quad C_n + X_n \rightsquigarrow NT \; x_1 \; \ldots \; x_p$$

We consider two cases :

1. All C_i are different (even up to renaming of variables).

   Let us show that the reduction chain considered is represented (up to renaming) in the reduction graph computed by Build$(C, \{C\}, T)$. Note that the type checking algorithm uses two renamings. The first one bounds the number of edges (i.e. makes the set CX finite); let us note it α_i (α_i is used implicitly in the definition of CX). The second one bounds the number of nodes; it is noted σ_i.

Starting from C_0, Build considers all the pairs (C', X') such that $C_0 + X' \leadsto C'$ up to renaming of fresh variables introduced by X'. So, it must be the case that a pair $(\alpha_1 X_0, \alpha_1 C_1)$ has been considered in the reduction $C_0 + \alpha_1 X_0 \leadsto \alpha_1 C_1$. But $\alpha_1 C_1$ might have been already present in the graph up to renaming. So in general, there is an edge $C_0 \overset{X'_0,\sigma_1}{\Leftrightarrow} C'_1$ in the graph such that $X'_0 = \alpha_1 X_0$ and $\sigma_1 C'_1 = \alpha_1 C_1$.

Using the same reasoning, we are ensured that the graph includes the edges

$$C'_1 \overset{X'_1,\sigma_2}{\Leftrightarrow} C'_2 \qquad \text{with } X'_1 = \alpha_2 \circ \sigma_1^{-1} \circ \alpha_1\ X_1$$

$$\ldots$$

$$C'_n \overset{X'_n,\sigma_{n+1}}{\Leftrightarrow} \{NT\ x_1 \ldots x_p\} \qquad \text{with } X'_n = \alpha_{n+1} \circ \sigma_n^{-1} \circ \ldots \circ \sigma_1^{-1} \circ \alpha_1\ X_1$$

Note that the domain of α_i comprises only fresh variables of X_{i-1} and that the domain of σ_j is included in the set of variables of C_j. Thus, if $j < i$ the domains of α_i and σ_j are disjoint and the X'_i can be rewritten as

$$X'_i = \sigma_i^{-1} \circ \ldots \circ \sigma_1^{-1} \circ \alpha_{i+1} \circ \ldots \circ \alpha_1\ X_i$$

Now, Check has verified that

$$A + X'_0 + \sigma_1 X'_1 + \ldots + \sigma_1 \circ \ldots \circ \sigma_n\ X'_n \overset{*}{\leadsto} \sigma_1 \circ \ldots \circ \sigma_{n+1}\ \{NTx_1 \ldots x_p\}$$

by replacing the X'_i's by their definition, we get

$$A + \alpha_1 X_0 + \alpha_2 \circ \alpha_1 X_1 + \ldots + \alpha_{n+1} \circ \ldots \circ \alpha_1\ X_n \overset{*}{\leadsto} \alpha_{n+1} \circ \ldots \circ \alpha_1\ \{NT\ x_1 \ldots x_p\}$$

The domains of α_i's are disjoint (they are renamings of *fresh* variables) so any composition of α_i's can be replaced by a unique variable renaming α whose domain is the set of fresh variables introduced by all the basic contexts. Furthermore, the domain of α is also disjoint from $Var(A)$ hence we have :

$$\alpha A + \alpha X_0 + \alpha X_1 + \ldots + \alpha X_n \overset{*}{\leadsto} \alpha\{NT\ x_1 \ldots x_p\}$$

which implies by lemma 1

$$A + X_0 + X_1 + \ldots + X_n \overset{*}{\leadsto} \{NT\ x_1 \ldots x_p\}$$

so $\quad M + A \overset{*}{\leadsto} M' + X_0 + \ldots + X_n + A \overset{*}{\leadsto} M' + \{NT\ x_1 \ldots x_p\} \overset{*}{\leadsto} \{T\}$

2. Otherwise, let C_j, C_k $(j < k)$ be the first residuals such that $\sigma C_j = C_k$.
Using the same reasoning as before we can show that there is a path in the graph

$$C_0 \overset{X'_0,\sigma_1}{\Leftrightarrow} C'_1 \ldots \overset{X'_{j-1},\sigma_j}{\Leftrightarrow} C'_j \ldots \overset{X'_{k-1},\sigma_k}{\Leftrightarrow} C'_k$$

$$\text{such that } X'_i = \sigma_i^{-1} \circ \ldots \circ \sigma_1^{-1} \circ \alpha\ X_i \text{ and } C'_i = \sigma_i^{-1} \circ \ldots \circ \sigma_1^{-1} \circ \alpha\ C_i$$

Since $\sigma C_j = C_k$ and C'_j and C'_k are renamings of respectively C_j and C_k, there is a renaming τ such that $\tau C'_j = C'_k$. This corresponds to a cycle in a graph such that C'_j leads to a nonterminal and is the first node belonging to a cycle on the path. So Check has verified that

$$A + X'_0 + \ldots + \sigma_1 \circ \ldots \circ \sigma_{j-1} X'_{j-1} \overset{*}{\leadsto} \sigma_1 \circ \ldots \sigma_j C'_j$$

According to the definition of X'_i and C'_i this can be rewritten as

$$\alpha A + \alpha X_0 + \ldots + \alpha X_{j-1} \overset{*}{\leadsto} \alpha C_j$$

thus, by lemma 1

$$A + X_0 + \ldots + X_{j-1} \overset{*}{\leadsto} C_j$$

and

$$M + A \overset{*}{\leadsto} M' + X_0 + \ldots + X_n + A \overset{*}{\leadsto} M' + C_j + X_j + \ldots X_n \overset{*}{\leadsto} M' + \{NTx_1 \ldots x_p\} \overset{*}{\leadsto} \{T\}$$

4.2 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

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

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

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

The type definition and associated *BigCrunch* rewriting system are :

$List$	$=$	$L\ x$	$L\ x$	$\leadsto$	$List$
$L\ x$	$=$	$\mathbf{succ}\ x\ y \ , \ L\ y$	$\mathbf{succ}\ x\ y \ , \ L\ y$	$\leadsto$	$L\ x$
$L\ x$	$=$	$\mathbf{end}\ x$	$\mathbf{end}\ x$	$\leadsto$	$L\ x$

The type checking amounts to the call

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

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

$$\mathbf{end}\ a \leadsto L\ a$$

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

$$\mathbf{end}\ a \overset{\emptyset,id}{\Leftrightarrow\!\!\to} L\ a$$

There is a single simple path and we are left with checking
 $\mathsf{Reduces\_to}$ $((\mathbf{succ}\ a\ b \ , \ \mathbf{end}\ b), L\ a, List)$
The set of possible residuals of $(\mathbf{succ}\ a\ b \ , \ \mathbf{end}\ b)$ is $\{(\mathbf{succ}\ a\ b \ , \ L\ b)\}$, so $\mathsf{Reduces\_to}$ $((\mathbf{succ}\ a\ b \ , \ L\ b),\ L\ a \ , \ List)$ is recursively called. The set of possible residuals of $(\mathbf{succ}\ a\ b \ , \ L\ b)$ by a $BigCrunch^{List}$ reduction is $\{L\ a\}$ and $\mathsf{Reduces\_to}$ $(L\ a, L\ a, List) = \mathsf{True}$. So, $\mathsf{TypeCheck}(Iota, List) = \mathsf{True}$ 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 *Listlast*.

$$Wrong : ListLast = \mathbf{succ}\ x\ z \ , \ \mathbf{last}\ x\ z \ , \ \mathbf{end}\ z \ \Rightarrow \mathbf{succ}\ x\ z \ , \ \mathbf{succ}\ z\ t \ , \ \mathbf{last}\ x\ t \ , \ \mathbf{last}\ z\ t \ , \ \mathbf{end}\ t$$

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 *Listlast* and its *BigCrunch* rewriting system are :

$Listlast$	$=$	$L\ x\ z$	$L\ x\ z$	$\leadsto$	$Listlast$
$L\ x\ z$	$=$	$\mathbf{succ}\ x\ y \ , \ \mathbf{last}\ x\ z \ , \ L\ y\ z$	$\mathbf{succ}\ x\ y \ , \ \mathbf{last}\ x\ z \ , \ L\ y\ z$	$\leadsto$	$L\ x\ z$
$L\ x\ z$	$=$	$\mathbf{succ}\ x\ z \ , \ \mathbf{last}\ x\ z \ , \ \mathbf{end}\ z$	$\mathbf{succ}\ x\ z \ , \ \mathbf{last}\ x\ z \ , \ \mathbf{end}\ z$	$\leadsto$	$L\ x\ z$

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

$$\mathbf{succ}\ x\ z \ , \ \mathbf{last}\ x\ z \ , \ \mathbf{end}\ z \ \leadsto L\ x\ z$$

but,

$$\mathbf{succ}\ x\ z \ , \ \mathbf{succ}\ z\ t \ , \ \mathbf{last}\ x\ t \ , \ \mathbf{last}\ z\ t \ , \ \mathbf{end}\ t \not\leadsto L\ x\ z$$

and the "Listlast" invariant is not maintained.

However, if we consider the insertion program:

$$Add : ListLast \ = \mathbf{succ}\ x\ y \ , \ \mathbf{last}\ x\ z \ \Rightarrow \mathbf{succ}\ x\ t \ , \ \mathbf{succ}\ t\ y \ , \ \mathbf{last}\ x\ z \ , \ \mathbf{last}\ t\ z$$

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

$$\mathbf{succ}\ x\ y\ ,\ \mathbf{last}\ x\ z\ ,\ L\ y\ z\ \rightsquigarrow L\ x\ z$$

and it is easy to check that

$$\mathbf{succ}\ x\ t\ ,\ \mathbf{succ}\ t\ y\ ,\ \mathbf{last}\ x\ z\ ,\ \mathbf{last}\ t\ z\ ,\ L\ y\ z \rightsquigarrow \mathbf{succ}\ x\ t\ ,\ \mathbf{last}\ x\ z\ ,\ L\ t\ z\ \rightsquigarrow\ L\ x\ z$$

and the "Listlast" invariant is maintained.

5 Refinement of Structured Gamma programs

The introduction put forward two main motivations for the design of Structured Gamma:

- Providing a notation leading to higher-level descriptions of programs manipulating data structures and making it possible to reason about this structure.
- Exposing relevant information to derive more efficient implementations.

The first issue was tackled in the previous sections. Here, we show how 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. Let us consider, as an illustration, the following "maximum segment sum" pure Gamma program.

$$\begin{array}{lcl}
maxss(M) & = & max_g(max_l(M)) \\
max_l & = & (i,v,s)\ ,\ (i',v',s')\ ,\ (i'=i+1)\ ,\ (s+v'>s')\ \Rightarrow\ (i,v,s)\ ,\ (i',v',s+v') \\
max_g & = & (i,v,s)\ ,\ (i',v',s')\ ,\ (s' \geq s)\ \Rightarrow\ (i',v',s')
\end{array}$$

The input parameter is a sequence of integers. A segment is a subsequence of consecutive elements and the sum of a segment is the sum of its values. The program returns the maximum segment sum of the initial sequence. The elements of the multiset are 3-tuples (i,v,s) where i is the position of value v in the sequence and s is the maximum sum (computed so far) of segments ending at position i. The s field of each 3-tuples is originally set to the v field. The program max_l computes local maxima and max_g returns the global maximum. The complexity of max_g is linear, even on a naïve implementation because any pair of elements (or its mirror) leads to a reaction and the action strictly decreases the size of the multiset. However the worst-case complexity of an unoptimised implementation of max_l is N^3, with N the size of the multiset. This cost is reached by a strategy choosing the first element (i,v,s) in decreasing order of i. As pointed out in [6], the order in which elements are selected is crucial indeed and most of the refinements leading to efficient optimisations of Gamma programs can be expressed as specific selection orderings. [6] introduces several refinements and 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. The basis 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. We first define the (semantic) notions of refinement on multisets, types and programs.

Definition 3 *Let R be a set of relation names, M, and M' multisets, T and T' types and P and P' Structured Gamma programs.*

- *The restriction of a multiset M' with respect to R is defined as*

$$M'/R\ =\ M' \Leftrightarrow \{\mathbf{r}\ a_1\ \ldots\ a_n \mid \mathbf{r} \in R\}.$$

- *M' is a R-refinement of M (noted $M'\ >_R\ M$) iff $M'/R\ =\ M$.*
- *T' is a R-refinement of T (noted $T'\ >_R\ T$) iff $M' : T'\ \Rightarrow (M'/R) : T$.*

- *P' is a partial R-refinement of P (noted $P' >_R P$) iff*

$$M' \overset{*}{\leftrightarrow}_{P'} N' \Rightarrow M'/R \overset{*}{\leftrightarrow}_P N'/R.$$

- *P' is a complete R-refinement of P (noted $P' \gg_R P$) iff*

$$M' \overset{*}{\leftrightarrow}_{P'} N' \Rightarrow M'/R \overset{*}{\leftrightarrow}_P N'/R.$$

The types *Doubly*, *Listlast* and *List*$_m$ defined in section 3 are refinements of the type *List* (with respect to $\{\mathbf{pred}\}$, $\{\mathbf{last}\}$ and $\{\mathbf{m1}, \mathbf{m2}\}$ respectively), but *Circular* is not a refinement of *List*. As an illustration of the relevance of this definition for deriving efficient implementations of Structured Gamma programs, let us consider yet another refinement of *List*:

$$\begin{array}{lcl} List_1 & = & L_1\ x \\ L_1\ x & = & \mathbf{succ}\ x\ y\ ,\ \mathbf{i}\ x\ ,\ L_1\ y \\ L_1\ x & = & L_2\ x \\ L_2\ x & = & \mathbf{succ}\ x\ y\ ,\ \mathbf{a}\ x\ ,\ L_2\ y \\ L_2\ x & = & \mathbf{end}\ x\ ,\ \mathbf{a}\ x \end{array}$$

It should be clear that *List*$_1$ is a R-refinement of *List* with $R = \{\mathbf{a}, \mathbf{i}\}$. We present now the translation of max_l in Structured Gamma and a new version max_{l1} which takes advantage of the extra relations to add restrictions on the control:

$max_l : List =$ $\mathbf{succ}\ x\ y\ ,\ (\overline{x}.s + \overline{y}.v > \overline{y}.s) \Rightarrow \mathbf{succ}\ x\ y\ ,\ y := (\overline{y}.v, \overline{x}.s + \overline{y}.v)$
$max_{l1} : List_1 =$
  $\mathbf{succ}\ x\ y\ ,\ \mathbf{i}\ x\ ,\ \mathbf{a}\ y,\ (\overline{x}.s + \overline{y}.v > \overline{y}.s) \Rightarrow \mathbf{succ}\ x\ y\ ,\ \mathbf{i}\ x\ ,\ \mathbf{i}\ y\ ,\ y := (\overline{y}.v, \overline{x}.s + \overline{y}.v)$
  $\mathbf{succ}\ x\ y\ ,\ \mathbf{i}\ x\ ,\ \mathbf{a}\ y,\ (\overline{x}.s + \overline{y}.v \leq \overline{y}.s) \Rightarrow \mathbf{succ}\ x\ y\ ,\ \mathbf{i}\ x\ ,\ \mathbf{i}\ y$

It can be shown that max_{l1} is a partial R-refinement of max_l with $R = \{\mathbf{a}, \mathbf{i}\}$. The intuition is that a program P' is a partial refinement of P if P can simulate all the "significant" reactions of P'[2]. It may be the case however that P' is not a proper implementation of P. The reason is that the termination condition for P' may be "stronger" than the termination condition of P (because of the extra relations). This situation occurs in the max_{l1} program if the initial multiset contains, say only $\mathbf{i}$ relations. In this example, $\mathbf{i}$ is the relation characterising inert elements (elements which cannot be modified by a reaction) and $\mathbf{a}$ corresponds to active elements. An extra condition has to be imposed to ensure that max_{l1} is a complete refinement of max_l. This condition is also expressed in terms of type refinements (roughly speaking, the initial multiset must be of type $List_2 = \mathbf{succ}\ x\ y\ ,\ \mathbf{i}\ x\ ,\ L_2\ y$). The following theorem shows that partial refinement can still serve as the basis of a correct program transformation:

Property 3 *If $P' >_R P$ then $M' \overset{*}{\leftrightarrow}_{P'} N'$ and $N'/R \overset{*}{\leftrightarrow}_P N \Rightarrow M'/R \overset{*}{\leftrightarrow}_P N$*

The proof of this theorem follows directly from the definition of partial refinement. The interesting consequence is that the property $P' >_R P$ allows us to "replace" P by the sequential composition $P \circ P'$ (with an intermediate conversion of the result N' of P' into N'/R).

In the above example, the complexity of the implementation of max_l in Structured Gamma is quadratic provided that the type *List* is implemented in memory as a standard linked list with pointers. So, the translation into Structured Gamma itself leads to a first improvement of the behaviour of the program. The complexity of max_{l1} is linear, but it is only a partial refinement of max_l and has to be composed with max_l for the transformation to be correct. If the initial multiset has the correct type *List*$_2$, then the result of max_{l1} is also a normal form for max_l and the execution of max_l is linear too: it amounts to check that a stable state has been reached. So partial refinement is strong enough to reduce the complexity to N^3 to $2N$ in this case. Proving complete refinement allows us to get rid of max_l and the resulting program is the expected one-pass linear walk through the list.

As a final comment, let us emphasize the fact that simple syntactic criteria can be used to check type and program (partial) refinement. Basically, a type T' is a R-refinement of type T if the definition of T can obtained (modulo renaming of nonterminals and cancelling useless rules) by removing from the definition of T' all the occurrences of $(\mathbf{r}, x_1, \ldots x_n)$ with $\mathbf{r} \in R$. The same idea applies to programs. These purely syntactic criteria can be used to check all the type and program refinements used in this section.

[2]The reactions which affect only relations in R are not significant for P

6 Conclusion

We have presented a way to structure multisets and Gamma programs. The main motivation was to express algorithms more elegantly as well as to provide means of refining programs for a more efficient implementation. The types introduced in this paper are context-free grammars. This makes the definition of square grids, for example, impossible. It is natural to investigate the extension to types as context-sensitive grammars. With such an extension, a square grid could be described as

$$\begin{array}{lcl}
Grid & = & E\ x\ y\ ,\ S\ x\ z\ ,\ \ L\ y\ z \\
L\ x\ y & = & E\ x\ z\ ,\ S\ y\ t\ ,\ \ L\ z\ t \\
L\ x\ y & = & \mathbf{end}\ x\ ,\ \mathbf{end}\ y \\
E\ x\ y\ ,\ S\ x\ z & = & \mathbf{east}\ x\ y\ ,\ \mathbf{south}\ x\ z\ ,\ E\ z\ t\ ,\ S\ y\ t \\
\mathbf{end}\ x\ ,\ E\ x\ y\_1\ ,\ \mathbf{end}\ z\ ,\ S\ z\ t\_1 & = & \mathbf{end}\ x\ ,\ \mathbf{east}\ x\ y\ ,\ \mathbf{end}\ z\ ,\ \mathbf{south}\ z\ t\ ,\ \mathbf{end}\ y\ ,\ \mathbf{end}\ t
\end{array}$$

The semantics of context-sensitive types is defined in the same way as the semantics of context-free types (section 2). The only difficulty lies in the checking process since context-sensitive *BigCrunch* reductions are not necessarily decreasing with respect to the size of the term. It may be possible to restrict type definitions such that a well-founded order can be found and our checking algorithm adapted. We are currently working on this issue

We think that the framework developed is of interest for a wider class of applications than Structured Gamma programs. The ability to define complicated graph structures concisely suggests domains such as the description of networks, software architecture or coordination languages.

In order to use Structured Gamma as a coordination language, we can interpret the addresses as individual entities to be coordinated. Their associated value defines their behaviour (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. As an illustration, a client-server architecture can be specified as follows in Structured Gamma:

$$\begin{array}{lcl}
CS & = & N\ n \\
N\ n & = & \mathbf{cr}\ c\ n\ ,\ \mathbf{ca}\ n\ c\ ,\ N\ n \\
N\ n & = & \mathbf{sr}\ n\ s\ ,\ \mathbf{sa}\ s\ n\ ,\ N\ n \\
N\ n & = & \mathbf{m}\ n
\end{array}$$

cr $c\ n$ and **ca** $n\ c$ denote respectively a communication link from a client c to the manager n (the client request channel), and the dual link from n to c (the client answer channel). The case for servers is similar. A correct program of type CS defines a valid dynamic transformation of the architecture. For instance:

$$\mathbf{m}\ n\ \Rightarrow\ \mathbf{m}\ n\ ,\ \mathbf{cr}\ c\ n\ ,\ \mathbf{ca}\ n\ c$$

is the addition of a new client.

The significant advantage of the use of Structured Gamma with respect to previous proposals for the formal definition of software architectures ([1, 9]) is that we consider the overall shape (or geometry) of the architecture as a first-class object. This allows us to check relevant properties of the architecture very easily (for instance, there is no direct communication link between a server and a client in the above architecture). In contrast [1], uses CSP programs to define the architecture, which leads to a description mixing the communication protocol with the geometry of the communication. We are currently studying the integration of the protocol specification (as separate values associated to the links) in Structured Gamma.

Another promising direction appears to be the extension of the usual data types of sequential programming languages. Our framework makes it easy to define common imperative structures (such as circular lists, doubly-linked lists,...). Type checking in this context would greatly contribute to the correctness of pointer-based programs. The graph types approach [10] 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, our types do not privilege any part of the graph and all operations on the structure appear explicitly in the rewrite rules. However, the application of our approach to imperative languages is not straightforward and needs further research.

Another related work is Raoult & Voisin's study of (hyper-)graph rewriting in a set-theoretic setting [12]. Their approach to graph rewriting is very close to ours: a hyper-graph is a set (a multiset in our case) of hyper-edges, noted $fx_1 \ldots x_n$, where f is a function symbol (a relation in our case) and $x_1 \ldots x_n$ are variables denoting vertices (addresses in our cases). They describe rewriting of sets of hyperedges and provide a criterion for confluence. The main departure of our work with respect to most of the previous studies of graph rewriting is the fact that we use graphs to represent data structures rather than programs. The underlying theory is not affected, but this specific point of view entails different kinds of problems (such as the type checking of section 4).

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

References

[1] R. Allen and D. Garlan. Formalizing architectural connection, in *Proc. 16th Int. Conf. Soft. Eng.*, IEEE Computer Society, pp. 71-80, 1994.

[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] M. Chaudron and E. de Jong, Towards a compositional method for coordinating Gamma programs, in *Proc. Coordination'96 Conference*, Cesena, 1996, Springer Verlag, LNCS 1061, pp. 107-123.

[5] D. Cohen and J. Muylaert-Filho, Introducing a calculus for higher-order multiset programming, in *Proc. Coordination'96 Conference*, Cesena, 1996, Springer Verlag, LNCS 1061, pp. 124-141.

[6] C. Creveuil. *Techniques d'analyse et de mise en œuvre des programmes Gamma*, Thesis, University of Rennes, 1991.

[7] C. Hankin, D. Le Métayer and D. Sands. A calculus of Gamma programs, in *Proc. of the 5th workshop on Languages and Compilers for Parallel Computing*, Yale, 1992, Springer Verlag, LNCS 757.

[8] C. Hankin, D. Le Métayer and D. Sands. A parallel programming style and its algebra of programs, in *Proc. of the PARLE conference*, Munich, 1993, Springer Verlag, LNCS 694, pp. 367-378.

[9] 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.

[10] N. Klarlund and M. Schwartzbach. Graph types. In *Proc. 20th Symp. on Princ. of Prog. Lang.*, pp. 196-205. ACM, 1993.

[11] D. Le Métayer, *Higher-order multiset programming*, in *Proc. of the DIMACS workshop on specifications of parallel algorithms*, American Mathematical Society, Dimacs series in Discrete Mathematics, Vol. 18, 1994.

[12] J.-C. Raoult and F. Voisin. Set-theoretic graph rewriting. *INRIA Research Report No 1665*, 1992.