Refinement of Chemical Programs using Strategies

Pascal Fradet
INRIA
Pascal.Fradet@inria.fr

Jean-Louis Giavitto
CNRS & IBISC
giavitto@ibisc.fr

Marnes Hoff
INRIA & IBISC
Marnes.Hoff@inria.fr

1 Introduction

The chemical reaction metaphor describes computation in terms of a chemical solution where molecules interact freely according to reaction rules. Chemical solutions are represented by multisets of elements and reactions by rewrite rules which consume and produce new elements according to conditions. In the Gamma formalism [3], a program is a collection of reaction rules made of a condition and an action. Execution proceeds iteratively by replacing elements satisfying the reaction condition by the elements specified by the action. The result of a Gamma program is obtained when a stable multiset is reached, that is to say, when no reaction can take place anymore.

The reaction *primes* below computes the prime numbers lower or equal to a given number N when applied to the multiset of all numbers from 2 to N:

$$primes = \textbf{replace}\ \ x,y\ \ \textbf{by}\ \ y\ \textbf{if}\ multiple(x,y)$$

where $multiple(x,y)$ is true if and only if x is a multiple of y. The execution is non deterministic and potentially highly parallel: if several disjoint tuples of elements satisfy the condition of one or several rules, the corresponding reactions can be performed in parallel.

The unconstrained data structure and reduction strategy permit to express algorithms without any artificial sequentiality. It confers a very high level nature to the language and makes Gamma suitable as an intermediate language in the program derivation process.

A major drawback is that a reasonably efficient implementation of the language is not straightforward. For example, a naive implementation for a single rule (like *primes*) consists in choosing a tuple of elements not yet processed and testing the reaction condition; if the condition is satisfied then the tuple is replaced by the result of the action; this process is iterated until no tuple can react. Such a blind approach of selection of elements and ordering of reactions is usually very inefficient. Assuming a multiset of size n, detecting the termination of a single k-ary reaction alone takes $O(n^k)$ steps. The sources of inefficiency are the selection of elements and the ordering of reactions which are left completely unspecified. Further, the lack of structured data makes it difficult to specify them. The approach sketched in this abstract aims at refining gamma programs by:

- structuring the multiset using a data type defining neighborhood relations between elements;
- describing the selection of elements according to their neighborhood and the previous selection;
- specifying the evaluation strategy (*i.e.,* the application of rules and termination).

If these three implementation aspects are written by the programmer using domain-specific languages, the final refined program is generated automatically. It consists in transforming the gamma program (representing the functionality) into a low-level program using the data structure, selection and strategy aspects. The crucial methodological advantage is that logical issues are decoupled from efficiency issues.

Due to space limitations, we do not present data refinement here. Our starting point is a Gamma program acting on an already structured multiset. That intermediate language is described in section 2. The refinement of control (*i.e.,* represented by the selection and strategy) and implementation issues are described in section 3 on a simple example. We conclude by outlining some related and future work.

2 Structured Gamma

After the data refinement step, programs act on a multiset equipped with a data structure. To express such programs, we use an extension of Gamma called *Structured Gamma* [8]. This extension is based on *structured multisets* which can be seen as a set of addresses satisfying specific relations and associated with a value. For example, the list $[5;\ 2;\ 7]$ is represented by a structured multiset whose set of addresses is $\{a_1,\ a_2,\ a_3\}$ and associated values (written $\overline{a_i}$) are $\overline{a_1} = 5$, $\overline{a_2} = 2$, $\overline{a_3} = 7$. Let begin and end be unary relations and next be a binary relation, the addresses satisfy

$$\mathsf{begin}\ a_1,\ \mathsf{next}\ a_1\ a_2,\ \mathsf{next}\ a_2\ a_3,\ \mathsf{end}\ a_3$$

A new notion of type is introduced in order to characterize precisely the structure of the multiset. A type is defined in terms of a context-free graph grammar [9]. A structured multiset belongs to a type T if its underlying set of addresses satisfies the invariant expressed by the grammar defining T. As an example, doubly-linked lists can be defined by the following context-free graph grammar:

$$\begin{array}{lcl} \mathit{Doubly} & = & \mathsf{begin}\ x,\ L\,x \\ L\,x & = & \mathsf{next}\ x\,y,\ \mathsf{prev}\ y\,x,\ L\,y \\ L\,x & = & \mathsf{end}\ x \end{array}$$

Any multiset which can be produced by this grammar belongs to the *Doubly* type. The variables in the rules are instantiated with addresses in the multiset. The non terminal $L\ x$ can be seen as standing for a doubly-linked list starting at address x. A singleton list will be represented as $\mathsf{begin}\,x, \mathsf{end}\,x$. This grammar ensures that next and prev are partial functions (*i.e.*, each address has at most a single next and prev address).

A reaction in Structured Gamma can test and modify the relations on addresses as well as the values of addresses. The following Structured Gamma programs manipulate doubly-linked lists. The reaction *iota* takes a singleton doubly-linked list $[n]$ and yields the doubly-linked list $[1;\ 2;\ldots;\ n]$ and *sort* sorts a doubly-linked list by exchanging ill-sorted adjacent elements.

$$\begin{array}{lcl} \mathit{iota} : \mathit{Doubly} & = & \textbf{replace}\ \mathsf{begin}\ a \quad \textbf{by}\quad \mathsf{begin}\ b,\ \mathsf{next}\ b\,a,\ \mathsf{prev}\ a\,b,\ b := \overline{a} - 1 \quad \textbf{if}\quad \overline{a} > 1 \\ \mathit{sort} : \mathit{Doubly} & = & \textbf{replace}\ \mathsf{next}\ a\,b \quad \textbf{by}\quad \mathsf{next}\ a\,b,\ (a,b) := (\overline{b},\overline{a}) \quad \textbf{if}\quad \overline{a} > \overline{b} \end{array}$$

Actions must also state explicitly how the relations (which can be seen as pointers) are modified. New addresses can be added to the multiset with their value and relations, like b in the *iota* reaction. On the other hand, a selected address which does not occur in the result of the action disappears from the multiset.

The formalism is expressive enough to define complicated pointer-like structures (*e.g.*, circular or skip lists, red-black trees, *etc.*.). Reactions can be statically checked according to the type definitions [8]. For example, type checking ensures that the above reactions preserve the *Doubly* type.

3 Refinement of Control through Strategies

The refinement of control is achieved by the specification of the selection of elements and the scheduling of rules. We present these steps on an example: the refinement of the previous *sort* program. We show how the same reaction rule can be refined in several sorting programs depending on selection and strategy. As the reaction *sort*, these programs operate by swapping ill-sorted adjacent elements.

A preliminary step is to parametrize the original reaction rule:

$$\mathit{sort}(a) = \ \textbf{replace}\ \mathsf{next}\ a\,b\ \textbf{by}\ \mathsf{next}\ a\,b,\ (a,b) := (\overline{b},\overline{a})\ \ \textbf{if}\ \ \overline{a} > \overline{b}$$

Parameters should be sufficient to select all the addresses involved by following relations/pointers (here next). Parameter a will serve to apply the rule to selected elements. The application of a parameterized reaction to its arguments is considered as *one-shot*. When the reaction *sort* is applied to an address, it either performs the swap or fails (depending on the condition $\overline{a} > \overline{b}$) then, in both cases, stops. It is a one-shot rule whereas the original *sort* reaction is n-shot (*i.e.,* it is applied until the multiset becomes stable). Considering the parameterized and one-shot version of reaction rules is crucial to control their application to specific (selected) elements as well as their application ordering.

Gnome sort We describe the selection and strategy languages corresponding to *gnome sort*, a simple exchange type of sort. The algorithm finds the first place where two adjacent elements are in the wrong order, and swaps them. It proceeds by taking into account that a swap can introduce a new out-of-order adjacent pair right before or after the two swapped elements.

The selection is described by the following functions:

$$\begin{array}{lcll} \mathit{Init} & = & x & \mid \ \mathsf{begin}\ x \\ \mathit{Test}(x) & = & & \mid \ \mathsf{begin}\ x \quad \textbf{or} \quad \mathsf{prev}\ x\ y\ ,\ \overline{x} \geq \overline{y} \\ \mathit{Succ}(x) & = & y & \mid \ \mathsf{next}\ x\ y \\ \mathit{Pred}(x) & = & y & \mid \ \mathsf{prev}\ x\ y \end{array}$$

A selection function takes zero or more address arguments and returns zero or more addresses as result. Like a reaction rule, a selection can succeed or fail depending of its condition (occurring after $\mid$). In our example, *Init* returns the address of the first element of a list; *Test* checks whether its parameter x is the first element of the list or is well-ordered compared to its predecessor; *Succ* (resp. *Pred*) returns the next (resp. previous) address in the list. If its condition is false, *Test* fails. Similarly, *Pred* will fail if its parameter is the first element of the list (*i.e.,* has no predecessor). Note that if a selection function may test relations and values, it cannot modify the multiset.

Our scheduling language was inspired from Chaudron's schedules [5]. A schedule is a collection of recursive functions made of calls to selection and one-shot reactions composed with sequential, parallel and conditional operators. A call to a selection function *Sel* is of the form $(a,\ldots) = \mathit{Sel}(b,\ldots)$; it binds the names $a,\ldots$ to the addresses returned by *Sel*. The conditional schedule (E **then** S_1 **else** S_2) executes the rule/selection E and proceeds with S_1 if it succeeds or S_2 if it fails. We write ($E \rightarrow S$) as a shorthand for (E **then** S **else** *skip*). If E succeeds it proceeds with the schedule S or terminates with the empty schedule *skip* otherwise. The sequential composition $S_1;S_2$ executes S_1 and proceeds with S_2 (even if S_1 is a reaction that failed). The schedule for gnome sort is:

$$\begin{array}{rcll} \mathit{Gnome} & = & a = \mathit{Init} \rightarrow G(a) & \\ G(a) & = & \mathit{Test}(a) \quad \textbf{then} & a' = \mathit{Succ}(a) \ \rightarrow \ G(a') \\ & & \textbf{else} & a' = \mathit{Pred}(a) \ \rightarrow \ \mathit{sort}(a');\ G(a') \end{array}$$

The schedule G is called with the address of the first element of the list. Initially, $\mathit{Test}(a)$ succeeds and the then-branch is executed. If there is a successor a' the schedule proceeds with that new address, otherwise the schedule ends. The predicate $\mathit{Test}(a)$ fails when a and its predecessor are ill-ordered. The schedule selects the predecessor, calls the reaction *sort* (which will swap values) and proceeds with the predecessor. So, gnome sort follows the next relation until it finds a ill-ordered element; it places it at its right place following the prev relation and then resume its traversal using next. It ends when *Succ* fails *i.e.,* when the end of the list is reached.

Note that if gnome sort follows n prev pointers to put a value at its place, it will have to traverse at least n next pointers before encountering new ill-ordered elements. An optimized version of gnome sort memorizes the current address when ill-ordered elements are first encountered and, when the element

has been put at its right place, proceeds directly from that address. This optimization is expressed by changing the schedule to:

$$\begin{array}{lcl} \mathit{Gnome} & = & a = \mathit{Init} \to G(a,a) \\ G(a,b) & = & \mathit{Test}(a) \quad \textbf{then} \quad a' = \mathit{Succ}(b) \ \to \ G(a',a') \\ & & \quad\ \ \textbf{else} \quad a' = \mathit{Pred}(a) \ \to \ \mathit{sort}(a');\ G(a',b) \end{array}$$

The schedule $G(a,b)$ now memorizes the address b to proceed after the value of a is put at its right place.

Reaction rules (here *sort*) remain the only mean to modify the multiset. The selection and schedule make the next elements to try and the termination explicit. There is no hidden implementation costs anymore. Here, we have chosen to specify deterministic and sequential strategies. Nevertheless, the schedule language is expressive enough to describe parallel and non deterministic strategies. The selection and schedules for two versions of bubble sort are described in the appendix.

Program generation After the specification of the base functionality by reactions, the data structures by a graph grammar and the control by a selection and schedule, the next step is to combine these components to generate a low-level, efficient C-like program. Unary relationships are implemented by global pointers. An address in a structured multiset is represented by a record, each field corresponding to a given binary relationship and pointing to the related address. With this representation, selection and reactions are translated into pointers dereferencing, predicate evaluation, address (de)allocation, expresion evaluation and assignments.

This compilation process is meant to be completely automatic. To this aim, several conditions must be checked beforehand: *(i)* the type can be represented by standard pointer structures; *(ii)* the parameterized reactions and selection functions can be translated directly into imperative commands; *(iii)* the schedule is deterministic and sequential.

Most of these conditions have already been considered in the context of Shape-C [7]. Shape-C was an extension of C with graph types and pointer manipulations expressed as Structured Gamma reactions. In particular, Shape-C had constraints to ensure that each reaction could be implemented as a constant-time C command. For example, the constraints on types enforce that relations are either unary or binary and that binary relations are functions. The constraints on reactions (and selection) enforce that all addresses involved can be obtained in constant time from the parameters of the reaction (resp. selection). These conditions, as well as the last one on schedules, can be checked using simple syntactic and static analyses.

4 Conclusion

With the growing complexity of software, programming would benefit from a clear separation of basic computations from their coordination. Functionality and correctness issues are expressed by the former while efficiency issues are addressed by the latter. We sketched an approach where Gamma is used to express the basic functionality of a program without artificial constraints on data structures or the control of execution. This enables the programmer to address efficiency issues separately in a second stage of the design process. Efficiency is achieved by adding structure to the multiset and by refining the chaotic behavior of Gamma into a sequential, deterministic one. In our simple examples, the basic functionality is represented by a single rule. As a consequence, the most complex part of the program lies in the scheduling. This would be more balanced with complex programs expressed as a collection of reaction rules.

Reflexivity (*e.g.,* in Maude [6] or the ρ_{bio}-calculus [1]) and strategies (*e.g.,* in Elan [4], TOM [2] or Stratego [11]) have been used to control the application of rules. In systems like Elan, elementary strategies (*e.g.,* applying a rewriting rule) are composed through `Sequence` or `Choice` operators. This

is the "classical approach of strategies" [10]. We got inspiration from a language proposed by Chaudron to improve the implementation of Gamma [5]. In that proposal, no data structure was considered and the selection of elements was encoded within the schedule language using integers. Furthermore, the goal was to improve the implementation not to produce a refined program. In our approach, refinement is expressed through data types, selection and schedule. Different refinements can be expressed using the same data type or selection functions. This kind of modularity is not possible in the classical approach of strategies. Selection and schedule enable the specification of sophisticated traversals of structured multisets which are much more closer to standard pointer structures than terms (*e.g.*, they can represent circular or doubly-linked lists).

We are currently working on the implementation of the automatic program generation process. Even if the main lines are clear, some analyses and the translation itself have not been completely formalized yet. As future work, we have to study the correctness issues of the approach. Clearly, all the steps (data refinement, selection and schedules) only serve to select a correct sequence of reactions among the set of possible executions. So, a partial correctness property can be shown to hold for all programs and refinements. However, since the schedule also encodes termination, it is not guaranteed that the refined program will terminate with a multiset stable for the initial rules (it may end prematurely). The correctness of this termination is likely to be too difficult to be proven in general. However, we hope to be able to generate the proof obligations needed to ensure total correctness. A longer term goal is to consider parallel strategies and the automatic generation of the corresponding parallel programs.

Acknowledgments This work is partially funded by the AutoChem ANR project.

References

[1] O. Andrei and H. Kirchner. Graph Rewriting and Strategies for Modeling Biochemical Networks. In *International Workshop on Natural Computing and Applications-NCA*, 2007.

[2] E. Balland, P. Brauner, R. Kopetz, P. Moreau, and A. Reilles. Tom: Piggybacking rewriting on java. *Lecture Notes in Computer Science*, 4533:36–47, 2007.

[3] J.-P. Banâtre and D. Le Métayer. Programming by multiset transformation. *Communications of the ACM*, 36(1):98–111, Jan. 1993.

[4] P. Borovanskỳ, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. *Electronic Notes in Theoretical Computer Science*, 4:169–189, 1996.

[5] M. Chaudron. Schedules for multiset transformer programs. In *Coordination Programming: Mechanisms, Models and Semantics*. Imperial College Press, 1996.

[6] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. *Electronic Notes in Theoretical Computer Science*, 4:126–148, 1996.

[7] P. Fradet and D. Le Métayer. Shape types. In *Proc. of Principles of Programming Languages*, Paris, France, Jan. 1997. ACM Press.

[8] P. Fradet and D. Le Métayer. Structured Gamma. *Science of Computer Prog.*, 31(2–3):263–289, 1998.

[9] J.-C. Raoult and F. Voisin. Set-theoretic graph rewriting. In *Proceedings of the International Workshop on Graph Transformations in Computer Science*, pages 312–325, London, UK, 1994. Springer-Verlag.

[10] E. Visser. A survey of rewriting strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, 57:109–143, 2001.

[11] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. *Rewriting Techniques and Applications (RTA'01), LNCS, volume 2051*, 2001.

Appendix

The same reaction *sort* can be refined in other kinds of exchange sort. These algorithms must, like the *sort* reaction, exchange only adjacent elements. Besides gnome sort, only two other classical algorithms satisfy this constraint: bubble sort and cocktail sort. Here, we describe the selection and strategies for two versions of bubble sort. Cocktail sort can be specified in a very similar fashion.

The selection for bubble sort uses the following functions:

$$\begin{array}{lcll} \mathit{OuterLoop}_0 & = & z & \mid \ \mathsf{begin}\ x,\ \mathsf{end}\ z,\ x \neq z \\ \mathit{OuterLoop}(z) & = & z & \mid \ \mathsf{begin}\ x,\ x \neq z \\ \mathit{InnerLoop}_0 & = & x & \mid \ \mathsf{begin}\ x \\ \mathit{InnerLoop}(x,z) & = & y & \mid \ \mathsf{next}\ x\ y,\ y \neq z \end{array}$$

The schedule for bubble sort is:

$$\begin{array}{rcl} \mathit{Bubble} & = & l = \mathit{OuterLoop}_0 \ \rightarrow\ a = \mathit{InnerLoop}_0 \ \rightarrow\ B(a,l) \\ B(a,l) & = & \mathit{sort}(a);\ a' = \mathit{InnerLoop}(a,l) \\ & & \textbf{then}\ \ B(a',l)\ \ \textbf{else}\ \ l' = \mathit{OuterLoop}(a) \ \rightarrow\ a' = \mathit{InnerLoop}_0 \ \rightarrow\ B(a',l') \end{array}$$

Bubble sort starts by selecting the first element (a) and the last one (l). The inner loop traverses the list from a to l swapping adjacent elements when they are ill-ordered. When a comes next to the limit (end) l (*i.e.,* $\mathit{InnerLoop}(a,l)$ fails) the process is repeated starting from the two first elements but setting the limit to its predecessor (a). The limit l represents the last element that still has to be considered in sorting. All elements after l are already sorted and in their final position. The process ends when the limit reaches the beginning of the list (*i.e.,* $\mathit{OuterLoop}(a)$ fails).

Bubble sort can be optimized by terminating as soon as a pass (an inner loop) does not entail any swap. In such a case, the list is known to be sorted and the program may stop. That optimization is performed by the following schedule (the selection functions remain the same):

$$\begin{array}{rcl} \mathit{Bubble} & = & l = \mathit{OuterLoop}_0 \ \rightarrow\ a = \mathit{InnerLoop}_0 \ \rightarrow\ B_0(a,l) \\ B_0(a,l) & = & \mathit{sort}(a)\ \ \textbf{then}\ \ B_1(a,l)\ \ \textbf{else}\ \ a' = \mathit{InnerLoop}(a,l) \ \rightarrow\ B_0(a',l) \\ B_1(a,l) & = & a' = \mathit{InnerLoop}(a,l) \quad \textbf{then}\ \ \mathit{sort}(a);\ B_1(a',l) \\ & & \qquad\qquad\qquad\qquad \textbf{else}\ \ l' = \mathit{OuterLoop}(a) \ \rightarrow\ a' = \mathit{InnerLoop}_0 \ \rightarrow\ B_0(a',l') \end{array}$$

The schedule work along the same lines as B except that it is made of two schedules B_0 and B_1. The schedule B_0 is used as long as a swap does not occur in the current inner loop and B_1 is used as soon as a swap occurs in the current inner loop. When an inner loop ends in B_0, it means that no swap occurred; the schedule just skips and terminates. When B_1 ends, the limit is changed and another inner loop is started with B_0.

Both versions of bubble sort operate on doubly-linked lists but, as they do not use the prev relation, they would also work with simply-linked lists. Starting from a more general reaction exchanging not only adjacent but any ill-sorted elements, other exchange sort algorithms (like quicksort) can also be obtained by refinement.