Generalized Multisets for Chemical Programming

Jean-Pierre Banâtre, Pascal Fradet, Yann Radenac

I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1762

GENERALIZED MULTISETS FOR CHEMICAL PROGRAMMING

JEAN-PIERRE BANÂTRE, PASCAL FRADET,
YANN RADENAC

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Generalized Multisets for Chemical Programming

Jean-Pierre Banâtre[*] , Pascal Fradet[**] , Yann Radenac[*]

Systèmes numériques — Systèmes communicants
Projets PARIS et POP-ART

Publication interne n ° 1762 — Novembre 2005 — 26 pages

Abstract: Gamma is a programming model where computation can be seen as chemical reactions between data represented as molecules floating in a chemical solution. This model can be formalized as associative, commutative, conditional rewritings of multisets where rewrite rules and multisets represent chemical reactions and solutions, respectively. In this article, we generalize the notion of multiset used by Gamma and present applications through various programming examples. First, multisets are generalized to include rewrite rules which become first-class citizen. This extension is formalized by the γ-calculus, a chemical model that summarizes in a few rules the essence of higher-order chemical programming. By extending the γ-calculus with constants, operators, types and expressive patterns, we build a higher-order chemical programming language called HOCL. Finally, multisets are further generalized by allowing elements to have infinite and negative multiplicities. Semantics, implementation and applications of this extension are considered.

Key-words: multisets, chemical programming model, rewriting, higher-order, infinite and negative multiplicities

(Résumé : tsvp)

[*] IRISA, Université Rennes 1, Inria Rennes
[**] Inria Rhône-Alpes

Centre National de la Recherche Scientifique
(UMR 6074) Université de Rennes 1 – Insa de Rennes

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

Multi-ensembles généralisés pour la programmation chimique

Résumé : Gamma est un modèle de programmation où les calculs peuvent être vus comme des réactions chimiques entre des données représentées comme des molécules flottant dans une solution chimique. Ce modèle peut être formalisé par la réécriture associative, commutative et conditionnelle de multi-ensembles où les règles de réécriture et les multi-ensembles représentent respectivement les réactions et les solutions chimiques. Dans cet article, nous généralisons la notion de multi-ensemble utilisé dans Gamma et nous présentons des applications à travers divers exemples de programmes. Dans un premier temps, les multi-ensembles sont étendus pour inclure les règles de réécriture qui deviennent donc des molécules comme les autres (ordre supérieur). Cette extension est formalisée par le γ-calcul, un modèle de calcul chimique qui contient en quelques règles l'essence même des programmes chimiques d'ordre supérieur. Ensuite, en étendant le γ-calcul avec des constantes, des opérateurs, des types et des motifs plus expressifs, nous construisons un langage de programmation qu'on appelle HOCL. Enfin, les multi-ensembles sont de nouveau étendus en permettant d'exprimer des multiplicités infinies et négatives. Sémantique, implémentation et applications de cette extension sont abordées.

Mots clés : multi-ensemble, modèle de programmation chimique, réécriture, ordre supérieur, multiplicités infinie et négative

1 Introduction

The Gamma formalism was proposed in (Banâtre & Le Métayer 1993) to capture the intuition of computation as the global evolution of a collection of atomic values interacting freely. Gamma 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 made of a *reaction condition* and an *action*. Execution proceeds 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 state is reached, that is to say, when no more reactions can take place.

For example, the computation of the maximum element of a non empty multiset can be described by the reaction rule

$$\mathbf{replace}\, x, y \,\mathbf{by}\, x \,\mathbf{if}\, x \geq y$$

meaning that any couple of elements x and y of the multiset is replaced by x if the condition is fulfilled. This process goes on till a stable state is reached, that is to say, when only the maximum element remains. Note that, in this definition, nothing is said about the order of evaluation of the comparisons. If several disjoint pairs of elements satisfy the condition, the reactions can be performed in parallel.

Gamma can be formalized as a multiset rewriting language. The literature about Gamma, as summarized in (Banâtre, et al. 2001), is based on finite multisets of basic values (often called bags). However, one may think of extensions to this basic concept by allowing elements of multisets to be reactions themselves (*higher-order multisets*), to have an infinite multiplicity (*infinite multisets*) and even to have a negative multiplicity (*hybrid multisets*).

In this paper, we investigate these unconventional multiset structures (higher-order, infinite and hybrid multisets) and show how they can be interpreted in a chemical programming framework. Section 2 presents the multiset as a mathematical structure and how it has been used to express programs. Section 3 presents the γ-calculus, a small higher-order calculus that summarizes the fundamental concepts of chemical programming. Section 4 introduces HOCL, a programming language built by extending the γ-calculus with constants, operators, types and more expressive patterns. Section 5 presents the extensions of HOCL needed to handle explicitly, positive or negative, finite or infinite, multiplicities. Section 6 proposes a representation of multisets suited to the implementation of these extensions. We conclude in Section 7 with a review of related work and a few perspectives.

2 A quick survey of the concept of multiset

2.1 Multiset as a mathematical structure

The notion of multiset is a concept appearing in many areas of mathematics and computer science. Intuitively, multiset are a generalization of sets in which elements can occur more than once. The number of occurrences of an element is called its *multiplicity*. The multiset $\{a, a, b\}$, which is not a set, is distinct from $\{a, b\}$; the multiplicity of a is 2 in the former and 1 in the later. The word multiset has been coined by De Bruijn in a private communication with D. Knuth (Knuth 1981);

other terms have appeared here and there in the literature such as bag, heap, sample, occurrence set, etc. A survey of the theory of multisets can be found in (Blizard 1991).

Even if the concept of multiset is very present in mathematics, logic and, more and more, in computer science, it has long been eclipsed by the classical Cantorian view of a set. Cantor states that a given element can appear only once in a set. We will not go into details here, but its is clear that this vision suffers some limitations; a well-known representation of numbers is a collection of units, for example the number 5 can be represented as $|\,|\,|\,|\,|$ which looks like a multiset of $|$. Cantor states that every $|$ is a different "instance" of $|$ and may be distinguished from the other $|$'s, however this looks a bit artificial. In some way, multiset can be considered as a non-classical (i.e. non-cantorian) set theory.

So far, we have implicitly assumed that the multiplicity of multiset elements was positive and finite. An intriguing extension of multisets, called hybrid multisets, has been introduced in (Loeb 1992). Hybrid multisets can contain elements with either positive or negative multiplicity. For example, the roots of the polynomial fraction $\frac{x-1}{(x-2)^2(x-3)}$ can be represented as the hybrid multiset $\{1, 2^{-2}, 3^{-1}\}$. Multiplicities of elements are denoted by the exponent; roots of polynomials under the fraction bar are denoted by a negative membership.

Finally, it is interesting to get rid of finiteness limitations and consider infinite multisets. Infinity can come from an infinity of different elements (e.g. all integers) or from an element with an infinite multiplicity. The later form, called *multiplet*, is the only form of infinity that we will consider here. For example, the multiplet $\{1^\infty\}$, represents a multiset containing an unbounded number of 1's.

2.2 Multiset as a programming structure

Coming back to Computer Science, the article of (Dershowitz & Manna 1979) introduced a multiset ordering and used it to prove program termination. Actually, given a well-founded ordering on elements of the multiset, it is possible to derive a well-founded ordering on multisets themselves. This nice result allows elegant proofs of termination which otherwise could be awkward. Without going into details, let us mention several areas of computer science where multisets are used: Petri nets, databases, logics, formal language theory, rewriting systems, etc. More can be found in (Calude, et al. 2001).

From Dershowitz & Manna's work, stemmed another fruitful idea (later called the Chemical Metaphor): the Gamma formalism where computation was presented as multiset rewriting (Banâtre & Le Métayer 1993). Gamma has been a source of inspiration in many unexpected areas as described in (Banâtre et al. 2001). Gamma is a simple model operating on multisets of basic data. A natural extension of Gamma is to generalize multisets so that they may contain not only data but also programs (abstractions/reactions). This is the first extension presented in this article in the form of a higher-order chemical calculus. We proceed by extending that simple model into an expressive Higher-Order Chemical Language: HOCL.

Another extension concerns infinite multiplets and their use in HOCL. These (infinite) multisets can be atomically handled as any element (for example, it is possible to atomically extract an infinite multiplet from a chemical solution), but it is also possible to select a finite subset of the (infinite)

multiset to react with other elements while leaving the multiplet unchanged (as it contains an infinity of elements!).

Finally, we consider the introduction of hybrid multisets and the interpretation of negative multiplicities in a programming context. There are several possible interpretations; the one we take consists in seeing an element with a negative multiplicity as an anti-element (an annihilator). For example, if an element such as 2^{-5} appears in a multiset it is interpreted as "annihilate" five 2's. Of course, the extensions can be combined to allow elements with a negative and infinite multiplicity. For example, $2^{-\infty}$ will instantaneously delete the 2's present or added in the multiset whatever their number of occurrences.

3 The γ-calculus: a higher-order chemical model

In this section, we introduce a higher-order chemical model called the γ-calculus (Banâtre, et al. 2005a, Banâtre, et al. 2005b). γ-expressions are made of molecules. A molecule can be (cf. Grammar 1) (1) a variable x that can represent any molecule, (2) a γ-abstraction $\gamma(P)\lfloor M_1 \rfloor.M_2$ where P is the pattern which determines the format (or type) of the expected molecule, M_1 is the reaction condition and M_2 the result of the reaction, (3) a compound molecule (M_1, M_2) built with the associative and commutative constructor ",", or (4) a solution denoted by $\langle M \rangle$ which isolates a molecule M from the others. The model is completed by a small pattern language to match multisets or solutions.

M	$::=$	x	*; variable*
	$\mid$	$\gamma(P)\lfloor M_1 \rfloor.M_2$	*; γ-abstraction (reaction rule)*
	$\mid$	M_1, M_2	*; multiset*
	$\mid$	$\langle M \rangle$	*; solution*
P	$::=$	x	*; matches any molecule*
	$\mid$	P_1, P_2	*; matches a compound molecule*
	$\mid$	$\langle P \rangle$	*; matches an inert solution*

Grammar 1: Syntax of molecules in the γ-calculus.

Molecules can be freely organized using the associativity and commutativity (AC) of the multiset constructor ",":

$$(M_1, M_2), M_3 \equiv M_1, (M_2, M_3) \qquad M_1, M_2 \equiv M_2, M_1$$

These rules can be seen as a formalization of the Brownian motion of chemical solutions. The operator $\equiv$ denotes the syntactic equality of two molecules. Two molecules are syntactically equal if any of them can be rewritten in the other one by the AC operations and by the renaming of bound variables.

A solution $\langle M \rangle$ is a membrane that encapsulates a molecule M. Molecules inside a solution cannot react or be rearranged with molecules outside that solution. However, molecules can be explicitly added to (or extracted from) solutions by reactions.

Another distinctive feature of chemical models is the reaction concept. In our model, it is represented by a conditional rewrite rule called the γ-reduction. In order to represent conditions, we assume an encoding for the booleans $\mathbf{true}$ and $\mathbf{false}$, for example:

$$\mathbf{true} \stackrel{\text{def}}{=} \gamma\langle x \rangle.\gamma\langle y \rangle.x \quad \text{and} \quad \mathbf{false} \stackrel{\text{def}}{=} \gamma\langle x \rangle.\gamma\langle y \rangle.y$$

A reaction is a rule of the form

$$(\gamma(P)\lfloor C \rfloor.M),\ N \rightarrow \phi M \qquad \text{if } match(P, N) = \phi \wedge \phi C \stackrel{*}{\rightarrow} \mathbf{true}$$

If a γ-abstraction "meets" a closed molecule N that matches the pattern P (modulo a substitution ϕ) and satisfies the reaction condition C (i.e. ϕC reduces to true), then they may react. The γ-abstraction $\gamma(P)\lfloor C \rfloor.M$ and the molecule N are replaced by the molecule ϕM (i.e. the body of the abstraction after substitution).

Substitution maps pattern variables to molecules e.g. $\phi = \{x \mapsto \mathbf{true}, y \mapsto \mathbf{false}\}$. A substitution is applied to a molecule using the following rules

$$\begin{array}{rcll} \phi x & = & M & \text{if } \phi = \{\ldots, x \mapsto M, \ldots\} \\ \phi\ (M_1, M_2) & = & (\phi M_1), (\phi M_2) & \\ \phi\langle M \rangle & = & \langle \phi M \rangle & \\ \phi(\gamma(P)\lfloor C \rfloor.M) & = & \gamma(P)\lfloor C \rfloor.\phi_{|P} M & \end{array}$$

where $\phi_{|P}$ is the substitution ϕ restricted to the variables that do not occur in P.

Pattern-matching can either succeed (it returns a substitution ϕ) or fail (it returns $\mathbf{fail}$). It is formalized as follows:

$$\begin{array}{lcll} match(x, M) & = & \{x \mapsto M\} & \\ match(\langle P \rangle, \langle M \rangle) & = & match(P, M) \wedge Inert(M) & \\ match((P_1, P_2), (M_1, M_2)) & = & (match(P_1, M_1)) \oplus (match(P_2, M_2)) & \\ match(P, M) & = & \mathbf{fail} & \text{otherwise} \end{array}$$

A variable matches any molecule, a pattern $\langle P \rangle$ matches any inert solution $\langle M \rangle$ (i.e. no reaction can take place within M) such that P matches M. This entails that a molecule can be extracted from its enclosing solution only when it has reached an inert state. It is an important restriction that permits to order (sequentialize) rewritings.

A pattern P_1, P_2 matches any compound molecule M_1, M_2 such that P_1 matches M_1, P_2 matches M_2 and the two substitutions are compatible. Since patterns are non linear, variables occurring in P_1 and P_2 must match identical molecules. The operator $\oplus$ is defined as follows:

$$\phi_2 \oplus \phi_1 = \begin{cases} \phi_2 \circ \phi_1 & \text{if } \nexists x.\phi_1 x = M_1 \wedge \phi_2 x = M_2 \wedge M_1 \not\equiv M_2 \\ \mathbf{fail} & \text{otherwise} \end{cases}$$

$$\mathbf{fail} \oplus x = x \oplus \mathbf{fail} = \mathbf{fail}$$

where the composition of compatible substitutions is such that $(\phi_2 \circ \phi_1)M \stackrel{\text{def}}{=} \phi_2(\phi_1 M)$.

An execution consists in γ-reductions ("chemical" reactions) until the solution representing the program becomes inert and no further rewriting is possible. Besides AC rules which can always been applied, there are two structural rules:

$$locality\ \frac{M_1 \to M_2}{M, M_1 \to M, M_2} \qquad solution\ \frac{M_1 \to M_2}{\langle M_1 \rangle \to \langle M_2 \rangle}$$

The $locality$ rule states that if a molecule M_1 can react then it can do so whatever its context M. The $solution$ rule states that reactions can occur within nested solutions.

This model of computation is intrinsically non-deterministic and parallel. As long as reactions involve disjoint molecules, they can take place simultaneously in a solution. Consider, for example, the solution $\langle(\gamma(x,y).x), \mathbf{true}, \mathbf{false}\rangle$, it may reduce to two distinct inert solutions ($\langle\mathbf{true}\rangle$ or $\langle\mathbf{false}\rangle$) depending on the application of AC rules and whether x will match $\mathbf{true}$ or $\mathbf{false}$.

The γ-calculus is quite expressive and can easily encode the λ-calculus. The following translation gives a possible encoding for the strict λ-calculus. The function $[\![\cdot]\!]$ takes a λ-term and returns its translation as a γ-term.

$$\begin{array}{ll} [\![x]\!] & \stackrel{\text{def}}{=} x \\ [\![\lambda x.E]\!] & \stackrel{\text{def}}{=} \gamma\langle x\rangle.[\![E]\!] \\ [\![E_1\ E_2]\!] & \stackrel{\text{def}}{=} \langle [\![E_1]\!]\rangle,\ (\gamma\langle f\rangle.f, \langle [\![E_2]\!]\rangle) \end{array}$$

The standard call-by-name λ-calculus can also be encoded but the translation is slightly more involved. As in the λ-calculus, recursion, integers, booleans, data structures, arithmetic, logical and comparison operators can be defined within the γ-calculus. We do not give their precise definitions here since they are similar to their definitions as λ-terms. From now on, we will give our examples assuming these extensions (a pair of molecules is written $M_1{:}M_2$).

Note that abstractions ($\gamma(P)\lfloor C\rfloor.M$) disappear in reactions: they are said to be *one-shot*. It is easy (using recursion) to define n-shot reactions which do not disappear after reacting. We write them $\mathbf{replace}\ P\ \mathbf{by}\ M\ \mathbf{if}\ C$ as in Gamma. Formally:

$$\mathbf{replace}\ P\ \mathbf{by}\ M\ \mathbf{if}\ C \quad \stackrel{\text{def}}{=} \quad \mathbf{let\ rec}\ f = \gamma(P)\lfloor C\rfloor.(M, f)\ \mathbf{in}\ f$$

For instance, the following program

$$\langle\langle 2\rangle, \langle 10\rangle, \langle 5\rangle, \langle 8\rangle, \langle 11\rangle, \langle 8\rangle, \mathbf{replace}\langle x\rangle, \langle y\rangle\ \mathbf{by}\langle x\rangle\ \mathbf{if}\ x \geq y\rangle$$

computes the maximum of a multiset of integers. The reaction rule does not disappear and reacts as long as there are at least two integers in the solution. The resulting inert solution is

$$\langle\langle 11\rangle, \mathbf{replace}\langle x\rangle, \langle y\rangle\ \mathbf{by}\langle x\rangle\ \mathbf{if}\ x \geq y\rangle$$

Note that each integer is inside a solution so that the reaction can match exactly two integers ($\mathbf{replace}\langle x\rangle, \langle y\rangle\ \mathbf{by}\ldots$ would match any, possibly compound, molecule). This encoding is made useless in the next section using types and the ability to match molecules of designated types.

$$
\begin{array}{l}
\mathrm{largestPrime10} = \\
\quad \textbf{let } sieve = \textbf{replace} \langle x \rangle, \langle y \rangle \textbf{ by } \langle x \rangle \textbf{ if } x \operatorname{div} y \textbf{ in} \\
\quad \textbf{let } max = \textbf{replace} \langle x \rangle, \langle y \rangle \textbf{ by } \langle x \rangle \textbf{ if } x \geq y \textbf{ in} \\
\quad \langle \langle \langle 2 \rangle, \ldots, \langle 10 \rangle, sieve \rangle, (\gamma \langle x \rangle . x, max) \rangle
\end{array}
$$

Program 1: Computes the largest prime number lower than 10.

Program 1 is the higher-order chemical version of the sieve of Eratosthenes used to compute the largest prime number lower than 10. The execution proceeds as follows:

$$
\begin{array}{rcl}
\langle \langle \langle 2 \rangle, \ldots, \langle 10 \rangle, sieve \rangle, \gamma \langle x \rangle . (x, max) \rangle & \stackrel{*}{\rightarrow} & \langle \langle \langle 7 \rangle, \langle 5 \rangle, \langle 3 \rangle, \langle 2 \rangle, sieve \rangle, \gamma \langle x \rangle . (x, max) \rangle \\
& \stackrel{*}{\rightarrow} & \langle \langle 7 \rangle, \langle 5 \rangle, \langle 3 \rangle, \langle 2 \rangle, sieve, max \rangle \\
& \stackrel{*}{\rightarrow} & \langle \langle 7 \rangle, sieve, max \rangle
\end{array}
$$

First, the n-shot reaction $sieve$ computes all prime numbers. It selects two integers x and y such that x divides y (so, y is not a prime number) and replaces them by x (i.e. removes y). Several $sieve$ reactions can take place in parallel as long as they involve different pairs of integers. When the sub-solution becomes inert (i.e. all prime numbers have been computed), the abstraction $\gamma \langle x \rangle . x, max$ extracts the inert solution and adds the prime numbers to the reaction max which computes their maximum. The final inert solution is $\langle \langle 7 \rangle, sieve, max \rangle$. The one-shot reaction rule $\gamma \langle \langle i \rangle, x \rangle . \langle i \rangle$ could be used to remove reactions and return only the integer as result.

4 HOCL: a higher-order chemical language

HOCL is a programming language based on the previous model extended with expressions, types, pairs, empty solutions and naming (see Grammar 2). Expressions consist in integer, boolean, string constants and associated operations. This extension, already used in the previous section, is very standard and does not need further explanation. We also reuse the notation $\textbf{replace} \ldots \textbf{by} \ldots \textbf{if} \ldots$ for n-shot (recursive) reaction rules. We present each other extension in turn.

4.1 Types

The functional core of HOCL (the expressions) is statically typed using standard types (see Grammar 2). We do not describe the typing rules which are the same as any (first-order) statically typed functional language. The chemical style of programming has been designed to be very flexible. In particular, solutions contain usually molecules of different types (e.g. reactions, integers, etc.). Therefore, it would not make sense to enforce homogeneous solutions and compound molecules are typed using the universal type $\star$. Any type is a subtype of $\star$ ($\forall T,\ T \preceq \star$). Types are particularly

Solutions

$S ::= \langle M \rangle$	*; solution*
$\mid \langle\rangle$	*; empty solution*

Molecules

$M ::= x$	*; variable*
$\mid M_1, M_2$	*; compound molecule*
$\mid A$	*; atom*

Atoms

$A ::= x$	*; variable*
$\mid [name =]\gamma(P)\lfloor V \rfloor.M$	*; one-shot reaction rule, possibly named*
$\mid S$	*; solution*
$\mid V$	*; basic value*
$\mid (A_1{:}A_2)$	*; pair*

Basic Values

$V ::= x \mid 0 \mid 1 \mid \ldots \mid V_1 + V_2 \mid -V_1 \mid \ldots$	*; integer, boolean and string expressions*
$\mid \mathbf{true} \mid \mathbf{false} \mid V_1 \wedge V_2 \mid \ldots$	
$\mid V_1 = V_2 \mid V_1 \leq V_2 \mid \ldots$	
$\mid$ "string" $\mid V_1 @ V_2 \mid \ldots$	

Patterns

$P ::= x{::}T$	*; matches any molecule of type T*
$\mid \omega$	*; matches any molecule even empty*
$\mid name = x$	*; matches a named reaction*
$\mid \langle P \rangle$	*; matches an inert solution*
$\mid (P_1{:}P_2)$	*; matches a pair*
$\mid P_1, P_2$	*; matches a compound molecule*

Types

$T ::= B$	*; basic type*
$\mid T_1 \times T_2$	*; product type*
$\mid \star$	*; universal type*

Basic Types

$B ::= \mathrm{Int} \mid \mathrm{Bool} \mid \mathrm{String}$

Grammar 2: Syntax of programs.

useful in patterns where they serve to select values. The associated pattern-matching rule is

$$match(x{::}T, N) = \{x \mapsto N\} \text{ if } Type(N) \preceq T$$

We make use of type inference to circumvent type annotations in patterns. For instance, we may write $\gamma(x)\lfloor V \rfloor.x + 1$ instead of $\gamma(x{::}\mathrm{Int})\lfloor V \rfloor.x + 1$ since the type of x can be statically inferred.

4.2 Pairs

This extension, denoted here by $A_1{:}A_2$, is very standard. Note that the elements of a pair are atoms and not multisets. Pairs of multisets would play a role similar to solutions by providing a way of isolating compound molecules from each other.

The rule for pattern-matching pairs is:

$$match((P_1{:}P_2), (N_1{:}N_2)) = \phi_1 \oplus \phi_2 \quad \text{if } match(P_1, N_1) = \phi_1 \wedge match(P_2, N_2) = \phi_2$$

4.3 Empty solutions

The notion of empty solution in HOCL comes from the pattern ω which can match any molecules even the "empty one" (introduced below). This pattern is very convenient to extract elements from a solution. For example, the following reaction extracts 1's from its solution argument.

$$rmunit = \mathbf{replace}\langle x, \omega\rangle \mathbf{by}\langle\omega\rangle \mathbf{if}\ x = 1$$

The pattern ω matches the rest of the solution which is returned as result. If the solution contains only a 1 then ω matches the empty molecule and the empty solution is returned:

$$rmunit, \langle 2, 1, 3\rangle \rightarrow \langle 2, 3\rangle \quad \text{and} \quad rmunit, \langle 1\rangle \rightarrow \langle\rangle$$

The rule for pattern-matching ω is just

$$match(\omega, M) = \{\omega \mapsto M\}$$

The "empty molecule" is introduced by the rule for patterns on the form (P, ω).

$$match((P, \omega), M) \quad = \quad \begin{cases} (match(P, M_1)) \oplus (match(\omega, M_2)) & \text{if } M = M_1, M_2 \\ (match(P, M)) \ \oplus (match(\omega, \emptyset)) & \text{otherwise} \end{cases}$$

When a molecule M is decomposed into M_1, M_2 to match a pattern P_1, P_2, one of M_1 or M_2 can now be the empty molecule $\emptyset$. Reaction rules involving ω patterns need a special treatment. Consider, for example, the reaction

$$(\mathbf{replace}\langle x, \omega\rangle \mathbf{by}\,\omega), \langle 1\rangle, 2$$

With the usual reduction rules, this molecule would reduce to $\emptyset, 2$ which is not a legal molecule. Only the empty solution is legal, so if a reaction produces the empty molecule it must be reduced as follows

$$\langle(\gamma(P)\lfloor C \rfloor.M), N, X\rangle \rightarrow \langle X\rangle \quad \text{if } match(P, N) = \phi \wedge \phi C \wedge \phi M = \emptyset$$

with X possibly empty. The reaction takes into account its enclosing solution and becomes global. When a reaction produces a non-empty molecule it is reduced locally as usual.

4.4 Naming

Reactions can be named (or tagged) using the syntax $name = \gamma(P) \ldots$. Note that if others atoms can be named using pairs e.g. $(name{:}a)$, it would not be appropriate to use pairs to tag reactions since they would not be able to react with other molecules anymore. Names are used to match and extract specific reactions. The rule for pattern-matching named reaction is

$$match((name = x), (name = N)) \ = \ \{x \mapsto N\}$$

We assume that when the **let** operator names a reaction, that name is kept in the solution:

$$\mathbf{let}\, name = M\, \mathbf{in}\, N \ \stackrel{\text{def}}{=} \ N[(name = M)/name]$$

that is, the occurrences of $name$ in N are replaced by $name = M$. For example, in the following example, the reaction incrementing the integer is named $succ$. After an arbitrary number of increments, the reaction $stop$ removes $succ$ from the solution:

$$\begin{array}{l}\mathbf{let}\, succ = \mathbf{replace}\, x\, \mathbf{by}\, x + 1\, \mathbf{in} \\ \mathbf{let}\, stop = \gamma((succ = x), \omega).\omega\, \mathbf{in} \\ \langle 1, succ, stop\rangle\end{array}$$

This example also illustrates non-determinism in HOCL since the resulting solution may be any integer.

4.5 Example of a distributed versions system (DVS)

As a more involved example, we consider several persons editing concurrently a document made out of a set of files. These editors are distributed over a network and each one works on one node of that network. Each node is independent from the others. Each editor makes his own modifications in the files and commits them locally on his node. So each editor keeps a local version (and its history) of these files. That version consists in the start files and several ordered patches applied to them: this history is called a *branch*. From time to time, two or more editors merge their branches so that an editor propagates (pushes) its modifications to others and/or get changes from other editors.

The following example is inspired from Monotone[1], a distributed version control system. Versions are identified by a hash ($Sha1$) which is used to check if two editors have the same version or not. Our system provides also the function $Merge(b_1, b_2)$ which returns a branch that contains all modifications from two given branches b_1 and b_2. If a conflict occurs, the initiator of the merge must resolve it. For simplicity sake, we assume in this example that the function $Merge$ always succeeds: either there is not any conflict, or if any conflict occurs it is solved by an editor.

An editor can express his dependency on modifications made by other editors. If the editor on node N_i depends on modifications made by editor on node N_j then the boolean function $Serve(b_i, b_j)$ will be true. In other words, modifications present in the branch B_i should be propagated to the branch b_j. They may be both dependent on each other. Since any branch can merge with

[1] `http://venge.net/monotone/`

$$
\begin{array}{l}
\mathrm{dvs} = \\
\quad \textbf{let}\, edit = \textbf{replace}\, b \,\textbf{by}\, Edit(b) \,\textbf{in} \\
\quad \textbf{let}\, push = \textbf{replace}\, b_1, b_2 \\
\qquad\qquad \textbf{by}\, b_1, Merge(b_1, b_2) \\
\qquad\qquad \textbf{if}\, Serve(b_1, b_2) \wedge Sha1(b_1) \neq Sha1(b_2) \\
\quad \textbf{in} \\
\quad \textbf{let}\, sync = \textbf{replace}\, b_1, b_2 \\
\qquad\qquad \textbf{by}\, Merge(b_1, b_2), Merge(b_2, b_1) \\
\qquad\qquad \textbf{if}\, Serve(b_1, b_2) \wedge Serve(b_2, b_1) \wedge Sha1(b_1) \neq Sha1(b_2) \\
\quad \textbf{in} \\
\quad \textbf{let}\, crash = \textbf{replace}\, b_1 \,\textbf{by}\, Start \,\textbf{if}\, Crash(b_1) \,\textbf{in} \\
\quad \textbf{let}\, freeze = \textbf{replace}\, (edit = e), x \,\textbf{by}\, x \,\textbf{in} \\
\quad \textbf{let}\, newVersion = \textbf{replace}\, \langle b_1, x\rangle \,\textbf{by}\, NewRelease(b_1), \langle b_1, edit, x\rangle \,\textbf{in} \\
\quad \langle\langle B_1, \ldots, B_n, edit, push, sync, crash, freeze\rangle, newVersion\rangle
\end{array}
$$

Program 2: Distributed Versions System.

any other branch, editors have to organize themselves so that all modifications from all editors are taken into account sooner or later. For example, the $Serve$ function may induce a tree where modifications may be propagated from the root to the leaves and vice versa. Or they may be organized as a ring, or any structures. Regularly, a freeze (snapshot) of the document is made to release a new version to users. This is performed by a call to the function $NewRelease$.

The overall system is described by Program 2. It consists in a solution containing all branches b_i. The reaction rule $edit$ represents the edition of any branch. It adds a modification to a branch, a call to the function $Edit$. Reaction rules $push$ and $sync$ merge branches: $push$ propagates modifications in one way, and $sync$ synchronizes two branches. If a node crashes ($Crash(b_i)$), the editor loses the corresponding branch. The reaction rule $crash$ resets the corresponding branch to an empty branch ($Start$). At any time, the reaction $freeze$ can initiate a snapshot of the document by removing the edition rule $edit$ to stop any modification. When the solution becomes inert, all branches linked by a $Serve$ relation are up to date and the reaction $newVersion$ can occur. It uses a branch that has all the modifications (it depends on the relations $Serve$) to release a new version (a call to $NewRelease$) and regenerates the system by adding the rule $edit$ to allow new modifications for the next release. Figure 1 gives a possible state reached by the system. The edition is pending and two releases have been made ($Version1$ and $Version2$).

This example illustrates several properties of HOCL:

- The execution is *non-deterministic*. Any two branches may react to merge their differences (if at least one of them serves the other). Merges (reactions $push$ and $sync$) may not occur each time a modification is made on a branch. In fact, editions and merges are asynchronous: several editions may occur before a merge.

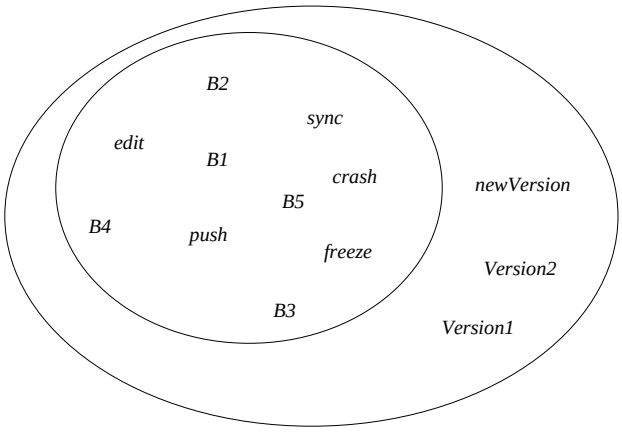

Figure 1: A possible state of the DVS.

- The execution is potentially *parallel*. Several editions may occur at the same time and several merges may happen at the same time if they deal with disjoint branches.
- The system is *autonomic* in that it is self-repairing. If a crash occurs, we lose a branch, but a simple $push$ or $sync$ with another branch allows to recover all modifications that have been propagated (however the editor loses all his local non-propagated modifications). Other autonomic properties may be included in a chemical program. The interested reader is referred to (Banâtre, et al. 2004) for more details on autonomic chemical programs.
- The specification is *higher-order* and manipulates reaction rules to express coordination. The $freeze$ reaction removes the $edit$ rule to stop edition. The $newVersion$ rule waits for inertia to call $NewRelease$ which illustrates a basic sequentiality coordination. The $newVersion$ rule relaunches also the system by re-generating the solution with the rule $edit$.

5 Multiplets, infinite and hybrid multisets

Another generalization is to extend the class of multisets to infinite multisets and elements with a negative multiplicity. The extension amounts to introducing operations to explicitly manipulate the finite or infinite, positive or negative, multiplicity of elements. The syntax of these extensions are summarized in Grammar 3.

5.1 Multiplets

A *multiplet* is a finite multiset of identical elements. This notion relies on an equality relation between elements. Considering multiplets of reactions would cause semantic problems as it would

$\mathbf{let}\ choose = \gamma\langle x{::}String, \omega\rangle.x\ \mathbf{in}$
$\mathbf{let}\ wheel = \langle$ "cherry", "lemon", "bell", "bar", "plum", "orange", "melon", "seven"$\rangle\ \mathbf{in}$
$\mathbf{let}\ win = \gamma(x{::}String)^3.$"Jackpot!"$\ \mathbf{in}$
$\langle wheel, wheel, wheel, choose, choose, choose, win\rangle$

Program 3: The Jackpot program.

require an equality relation between programs whereas multiplets of solutions would pose implementations issues. In this paper, we limit ourselves to multiplets of basic values (integers, booleans, strings). In HOCL multiplets are defined and matched using an exponential notation (see Grammar 3):

- if v is a basic value then v^k $(k > 0)$ denotes a multiplet of k elements v. Similarly, in a reaction, x^k denotes a multiplet of k elements. The variable x must have a basic type ($x{::}B$).
- in order to match multiplets, the language of patterns is extended likewise. A pattern, P^k matches any multiplet of k identical elements matching P.

Semantically, a multiplet v^k is just a shorthand for k identical v's. Formally:

$$v^1 \stackrel{\text{def}}{=} v \quad \text{and} \quad v^k \stackrel{\text{def}}{=} v^{(k-1)}, v \quad \text{if } k > 1$$

Semantically, a pattern P^k is just a shorthand for the nonlinear pattern defined by

$$P^1 \stackrel{\text{def}}{=} P \quad \text{and} \quad P^k \stackrel{\text{def}}{=} P^{k-1}, P \quad \text{if } k > 1$$

For example, the reaction replacing four 1's by four 2's can be specified as

$$\gamma(x^4)\lfloor x = 1\rfloor.2^4 \quad \text{or equivalently} \quad \gamma(x,x,x,x)\lfloor x = 1\rfloor.2,2,2,2$$

Another elementary example is the n-shot reaction computing the root set of a multiplet by removing repeatedly pairs of identical elements:

$$toSet1 = \mathbf{replace}\, x^2\, \mathbf{by}\, x$$

In the Jackpot! program (Program 3), the reactions $choose$ pick up nondeterministically an element from the solutions representing the three wheels of a slot machine. The win reaction checks if the three drawn symbols are identical, i.e. if it can match a multiplet of size 3.

5.2 Variable-sized multiplets

A first generalization of multiplets is to allow variables in the exponentiation of constants or patterns. The size of a multiplet becomes dynamic.

Molecules

$M ::= \ldots$ *; as before*

$\quad | \quad V_1^{V_2}$ *; multiplet with an integer expression V*

Basic Values

$V ::= \ldots$ *; as before*

$\quad | \quad [-]\infty$ *; positive and negative infinity*

Patterns

$P ::= \ldots$ *; as before*

$\quad | \quad P^k$ *; matches a finite size multiplet*

$\quad | \quad P^x$ *; matches multiplets of any size*

$\quad | \quad P^{\overline{x}}$ *; matches all elements of a multiplet*

Grammar 3: HOCL extended with multiplets, infinite and negative multiplicities.

Let v be a basic constant and V an integer expression, then v^V denotes a multiplet. If the normal form of V is the integer k then $v^V \equiv v^k$. We assume in this section that $k > 0$. If $k = 0$ the multiplet is empty and is treated in much the same way as a ω-variable which has matched the empty molecule (cf. Section 4.3). The case of a negative exponent is dealt with in Section 5.4.

A pattern P^x matches any strictly positive number of identical atoms. Formally:

$$match(P^x, (V_1, \ldots, V_k)) \quad = \quad match(P, V_1) \oplus \{x \mapsto k\} \quad \text{if } k > 0 \wedge \forall i, j \in [1, k].V_i = V_j$$

The substitution returned by a successful match maps the exponent variable x to the number of matched values.

For example, the n-shot reaction computing the root set of a multiplet of the previous section can be expressed using variable sized multiplet matching:

$$toSet2 = \mathbf{replace}\, x^n \,\mathbf{by}\, x \,\mathbf{if}\, n > 1$$

Whereas the previous version ($toSet1$) eliminated duplicates two by two, the rule $toSet2$ eliminates a variable number (potentially greater than 2) duplicates at each step.

Another example is a quite natural specification of integer division (see Program 4). The program makes use of two values R and Q which can be distinct strings for example. The dividend x is translated into the multiplet R^x whereas the divisor d is left as an integer. The integer division of x by d is performed by grouping d occurrences of R's and replacing them by one occurrence of Q. When the solution becomes inert, the multiplicity of Q represents the quotient and the multiplicity of R represents the remainder. For example, the division of 7 by 2 is performed as follows:

$$\langle cluster, 2, R^7 \rangle \rightarrow \langle cluster, 2, R^5, Q \rangle \rightarrow \langle cluster, 2, R^3, Q^2 \rangle \rightarrow \langle cluster, 2, R, Q^3 \rangle$$

$$\begin{array}{l}\mathrm{intdiv} = \gamma(x{:}d).\\ \quad \mathbf{let}\ cluster = \mathbf{replace}\ (d, R^d)\ \mathbf{by} (d, Q)\ \mathbf{in}\\ \quad \langle R^x, d, cluster\rangle\end{array}$$

Program 4: Integer division.

5.3 Infinite multiplets

Another generalization consists in infinite multiplets. Let v be a basic value or a variable with a basic type, then v^∞ denotes an infinite multiplet made of an infinity of copies of v. Formally:

$$v^\infty \stackrel{\text{def}}{=} M \text{ such that } Card(M) = \infty \wedge \forall x \in M.\ x = v$$

We do not introduce patterns of the form P^∞ to match an infinity of identical elements. Indeed, extracting an infinity of elements from an infinity would not be well defined. Instead we introduce a pattern matching all occurrences of a constant in the solution. Using such patterns, infinite multiplets can be manipulated as a single atomic molecule.

The pattern $P^{\overline{x}}$ matches all identical atoms occurring in the enclosing solution. Formally:

$$\begin{array}{rcll} match_M(P^{\overline{x}}, N) & = & (match(P, a)) \oplus \{x \mapsto Card(N)\} & \wedge\ (\forall a' \in N. a' = a)\\ & & & \wedge\ a \notin M\end{array}$$

The substitution returned by a successful match maps the variable x to the finite or infinite multiplicity of the matched value.

Note that pattern matching must take an additional argument (here M) representing the remaining of the enclosing solution to check that all occurrences have been taken into account. The reduction of a reaction with such patterns is of the form:

$$\langle(\gamma(P)\lfloor C\rfloor.X), N, M\rangle \rightarrow \langle \phi X, M\rangle \qquad \text{if } match_M(P, N) = \phi \wedge \phi C$$

The complete solution is taken by the reaction and no other reaction in the same solution may occur in parallel. Taking atomically all identical elements of a solution is intrinsically a global operation.

For example, the n-shot reaction computing the root set of a multiplet of the previous sections can now be expressed as follows:

$$toSet3 = \mathbf{replace}\ x^{\overline{n}}\ \mathbf{by}\ x\ \mathbf{if}\ n > 1$$

All duplicates of an element are removed in one reaction rule. For example, the solution $\langle a^{10}, b^4, toSet3\rangle$ is rewritten in two steps:

$$\langle a^{10}, b^4, toSet3\rangle \rightarrow \langle a, b^4, toSet3\rangle \rightarrow \langle a, b, toSet3\rangle$$

As another example, consider the traditional quicksort program where a set of integers has to be compared with a predefined pivot. In order to distinguish the pivot from the other integers, we

assume that the pivot had a special type $Pivot$ (e.g. a type synonym of Int). In the following solution all integers lower or equal to the pivot are removed. We consider the pivot as a infinite multiplet of an integer of type $Pivot$ (5^∞ here):

$$\langle 5^\infty, 8, 3, 6, 4, 5, 3, \mathbf{replace}(p{::}Pivot), x, \omega \, \mathbf{by} \, \omega \, \mathbf{if} \, x \leq p \rangle$$

As the number of pivots is infinite, all possible reactions may be carried out independently. This is a way of expressing the fact that the pivot is a read only element and as such can be accessed concurrently. The use of read only elements in chemical specifications has been proposed in (Chaudron 1994).

A standard way of accommodating infinite objects in programming (e.g. in lazy functional languages) is to use generators and on-demand evaluation. Following this idea, we would represent infinite multiplets, for example the multiplet 4^∞, as

$$gen4 = \mathbf{replace} \, x \, \mathbf{by} \, x, x \, \mathbf{if} (x = 4)$$

However, this encoding suffers two main problems: it makes difficult to manipulate infinite multiplets (e.g. removing them) and the property $v^\infty, v^\infty \equiv v^\infty$ is not satisfied.

5.4 Negative multiplicities

Hybrid multisets (Blizard 1990, Loeb 1992) are a generalization of multisets where the multiplicity of elements can be negative. A molecule v^{-1} can be viewed as a piece of "antimatter"or an anti-v. Positive and negative multiplets of the same value cannot cohabit in the same solution, they merge into one multiplet whose exponent is the sum of their exponent. Assuming a representation of negative values v^{-1}, a negative multiplet is defined as:

$$v^{-k} \stackrel{\text{def}}{=} v^{-k+1}, v^{-1} \quad \text{if } -k < -1$$

The pattern P^{-1} is defined as matching (the representation of) v^{-1} such that $match(P, v)$. The pattern P^{-k} is defined as k occurrences of P^{-1} :

$$P^{-k} \stackrel{\text{def}}{=} P^{-k+1}, P^{-1} \quad \text{if } -k < -1$$

The representation of negative values using reaction rules consuming elements such as $\text{kill}_v = \gamma(x, \omega)\lfloor x = v \rfloor.\omega$ would not be sufficient. The intended semantics enforces that v and v^{-1} cannot be in a solution at the same time. There is no guarantee about when a reaction kill_v will react. When negative multiplicities are allowed, the negative and positive multiplets of the identical elements must be merged after each reaction before proceeding with other reactions. In other words, reactions become global rewritings w.r.t. their solution. We define this merging process using the new reduction relation $\hookrightarrow$ defined by the two following rules:

$$\langle v, v^{-1} \rangle \hookrightarrow \langle \rangle \qquad \langle v, v^{-1}, X \rangle \hookrightarrow \langle X \rangle$$

The rule for reactions becomes

$$\frac{match(P,N) = \phi \wedge \phi C \wedge \langle \phi M, X\rangle \overset{*}{\hookrightarrow} \langle Y\rangle \wedge \langle Y\rangle \not\hookrightarrow \langle Z\rangle}{\langle (\gamma(P)\lfloor C\rfloor.M), N, X\rangle \to \langle Y\rangle}$$

A reaction takes a molecule (N) matching its pattern but also the remaining of the solution (X). The positive and negative multiplets occurring in the result of the reaction (ϕM) are simplified with the other multiplets occurring in X. A reduction step is global and consists in a reaction followed by a normalization by $\hookrightarrow$.

Variable sized and infinite multiplet with negative multiplicities are defined exactly the same way as before. We match and produce values v^{-1} instead of v. In any case, a solution must be normalized using $\hookrightarrow$ between two successive reactions.

As an example of use of negative multiplicities, rational numbers $\frac{p}{q}$ are represented by a molecule which contains the prime factorization of p and q but with negative multiplicities for the latter. For example, $\frac{20}{9}$ is represented by the molecule $\langle 2^2, 5, 3^{-2}\rangle$. The product of rational numbers is computed simply by putting them in the same solution. For example, the product $\frac{20}{9} * \frac{15}{8}$ is performed by merging their representations:

$$\langle 2^2, 5, 3^{-2}\rangle, \langle 3, 5, 2^{-3}\rangle, \gamma(\langle f\rangle, \langle g\rangle).\langle f, g\rangle \to \langle 5^2, 3^{-1}, 2^{-1}\rangle$$

Infinite negative multiplets can be used to filter out all occurrences of an element (present or to come) within a solution. Let pi be the reaction computing the product of a multiset of integers. Then, the integer 1, being the neutral element of the product, can be deleted prior to performing pi. The pi operator may be encoded by:

$$pi = \gamma\langle x\rangle.\langle 1^{-\infty}, x, (\mathbf{replace}\, x, y\, \mathbf{by}\, x * y)\rangle$$

Before considering any product, all 1's are annihilated, for example:

$$\langle 2^2, 9, 1^3, 5, 6\rangle, pi \to \langle 1^{-\infty}, 2^2, 9, 5, 6, (\mathbf{replace}\, x, y\, \mathbf{by}\, x * y)\rangle \to \ldots$$

After stabilization, $1^{-\infty}$ must be replaced by 1 (in case that the solution contained only 1's) and then the reaction rule can be removed.

Other examples that come to mind include the specifications of a garbage collector that destroys useless molecules by generating their negative counterpart, or an anti-virus that generates $v^{-\infty}$ each time it identifies a virus v. The negative multiplet will remove all occurrences (present or future) of the corresponding virus from the solution.

6 Operational semantics and implementation

In previous work on chemical programming (Banâtre & Le Métayer 1993, Banâtre et al. 2001, Banâtre et al. 2005b), solutions were always represented straightforwardly as multisets of elements and reactions as AC rewritings. In the previous section, we followed the same idea and presented

the semantics of multiplets by enumerating them in order to keep using plain multisets and rewritings. However, we had to use infinite multisets and auxiliary reduction rules. Part of that semantics description (in particular, the treatment of infinite multisets) is not directly implementable whereas others facets (e.g. normalization by $\hookrightarrow$) seem very costly.

Here, we propose an alternative and more concrete (operational) semantics which can be used as a basis for a reasonable implementation of multiplets. Since our extensions can be seen as programming constructs manipulating the multiplicities of values, we propose a representation that makes multiplicities explicit.

6.1 Representation of solutions

The central idea is to use the standard mathematical representation of a multiset, that is, a function associating to each element of the multiset its multiplicity. Such a function can be represented by a table whose entries are the atoms of the solution; basic values are associated with a non-zero integer whereas other atoms (reactions, sub-solutions) are always associated with 1. In this paper, we represent such functions/tables by sets of indexed elements. A closed molecule M is represented by a set denoted by $[M]$.

$$\begin{array}{lll} [M] & ::= & [A] \mid [M_1], [M_2] \\ [A] & ::= & v^k \mid \gamma(P)\lfloor C \rfloor.M \mid \langle [M] \rangle \mid (A_1{:}A_2) \end{array}$$

Each basic value is associated with its multiplicities, other atoms are implicitly associated with 1 and sub-solutions are represented by a set as well. As before, atoms of the form a^1 are written a. The key property of that representation is for any basic values v_1 and v_2

$$v_1^{k_1}, v_2^{k_2} \in [M] \Rightarrow v_1 \neq v_2$$

that is to say, $[M]$ is a set w.r.t. to basic values.

Note that in set representation, a molecule X belongs to another one M (i.e. $X \in M$) if X appears with exactly the same multiplicities in M (modulo AC). For example:

$$2^3 \in (4, 2^3, 5) \qquad (2^3, 4) \in (4, 2^3, 5) \qquad \text{but} \qquad 2^2 \notin (4, 2^3, 5)$$

The translation of a closed molecule in its set-representation is described in Figure 2. The first rule (associativity) serves to transform the molecule into the normalized form $a_1, (a_2, \ldots, (a_{n-1}, a_n) \ldots)$. Identical basic values are merged into a single value associated with its global multiplicity. Sub-solutions are translated into set-representation as well whereas other atoms are left unchanged. Multiplicities can be negative and infinite, that is:

$$k_i \in \mathbb{Z}^*_\infty = -\infty, \ldots, -2, -1, 1, 2, \ldots, \infty$$

Addition is extended to deal with infinity as follows:

$$\forall k \in \mathbb{Z} \quad \infty + k = \infty \quad -\infty + k = -\infty \quad -\infty + \infty = \bot$$

For example, the solution $\langle 2, (\gamma(x).x + 1), \langle 2^3, 9, 2^{-\infty} \rangle, 2, (\gamma(y).y + 1) \rangle$ is represented as the set $\langle 2^2, (\gamma(x).x + 1), (\gamma(y).y + 1), \langle 2^{-\infty}, 9 \rangle \rangle$.

$$
\begin{array}{lcll}
[(M_1, M_2), M_3] & = & [M_1, (M_2, M_3)] & \\
[v^{k_1}, M] & = & \left\{ \begin{array}{ll} [v^{k_1+k_2}, M - v^{k_2}] & \text{if } v^{k_2} \in M \wedge k_1 + k_2 \neq 0 \\ [M - v^{k_2}] & \text{if } v^{k_2} \in M \wedge k_1 + k_2 = 0 \\ v^{k_1}, [M] & \text{if } \nexists v^{k_2} \in M \end{array} \right. & \\
[\langle M_1 \rangle, M_2] & = & [\langle M_1 \rangle], [M_2] & \\
[a, M] & = & a, [M] & \text{for other atoms } a \text{ i.e. reactions or pairs} \\
[\langle\rangle] & = & \langle\rangle & \text{empty solution} \\
[\langle M \rangle] & = & \langle [M] \rangle & \\
[v^k] & = & v^k &
\end{array}
$$

Figure 2: Transforming molecules into set representation.

6.2 Reduction of molecules in set representation

Pattern-matching a molecule in set-representation is more complex than before. It takes the complete solution and yields a substitution and a remainder. The remainder of a match is the molecule taken in entry minus the extracted molecule matching the pattern. For example, the only possible result for

$$match_{[]}((x^{\overline{y}}, z^2, z^2), (2^3, 4^5, \langle 7 \rangle))$$

is $(\{x \mapsto 2, y \mapsto 3, z \mapsto 4\}, (4, \langle 7 \rangle))$. The only molecule included in $(2^3, 4^5, \langle 7 \rangle)$ matching the pattern is $(2^3, 4^4)$.

We say that a molecule X is included into another molecule M, and we write $X \sqsubseteq M$, if it can be extracted from M. All atoms of X must occur in M with greater or equal multiplicities. For example, $2^2 \sqsubseteq (4, 2^3, 5)$ and, of course, $X \in M \Rightarrow X \sqsubseteq M$. By convention the empty molecule cannot be extracted from a molecule.

Pattern-matching is defined using this notion in Figure 3. A composed pattern P_1, P_2 is matched by considering P_1 and P_2 in sequence. Pattern-matching P_1 yields a substitution ϕ_1 and a remainder M_1. Next P_2 is matched against M_1 and yields a substitution ϕ_2 and a remainder M_2 (M_1 minus the extracted match). The result is the composition of the two substitutions and M_2.

The pattern $name = x$ matches any reaction of M tagged with that name.

The pattern $\langle P \rangle$ involves extracting a solution $\langle X \rangle$ from M and enforcing that P matches *completely* X (i.e. $match_{[]}(P, X)$ returns an empty remainder).

The pattern $x{::}T$ matches any molecule X whose type is smaller than T and which can be extracted from M. Similarly, the pattern ω matches any molecule which can be extracted from M but matches also the empty molecule.

The pattern $(P_1{:}P_2)$ involves extracting a pair from M and using standard pattern matching.

Patterns for multiplets involves selecting a basic value v^j from M. To match P^k (i.e. to extract v^k), j must greater or equal if positive (lower or equal if negative). Matching P^x amounts to ex-

$$
\begin{array}{lcl}
match_{[]}((P_1,P_2),M) & = & (\phi_1 \oplus \phi_2,\ M_2) \\
 & & \quad \text{if } match_{[]}(P_1,M) = (\phi_1,M_1) \wedge match_{[]}(P_2,M_1) = (\phi_2,M_2) \\
match_{[]}(x{::}T,M) & = & (\{x \mapsto X\},\ M - X) \\
 & & \quad \text{if } X \sqsubseteq M \wedge Type(X) \preceq T \\
match_{[]}(\omega,M) & = & \begin{cases} (\{x \mapsto \emptyset\},\ \emptyset) \\ \quad \text{if } M = \emptyset \\ (\{x \mapsto X\},\ M - X) \\ \quad \text{if } X \sqsubseteq M \end{cases} \\
match_{[]}(name = x,M) & = & (\{x \mapsto R\},\ M - (name = R)) \\
 & & \quad \text{if } (name = R) \in M \\
match_{[]}((P_1{:}P_2),M) & = & (\phi_1 \oplus \phi_2,\ M - (X_1{:}X_2)) \\
 & & \quad \text{if } match(P_1,X_1) = \phi_1 \wedge match(P_2,X_2) = \phi_2 \wedge (X_1{:}X_2) \in M \\
match_{[]}(\langle P \rangle,M) & = & (\phi,\ M - \langle X \rangle) \\
 & & \quad \text{if } \langle X \rangle \in M \wedge match_{[]}(P,X) = (\phi,\emptyset) \\
match_{[]}(P^k,M) & = & (match(P,v),\ M - v^j + v^{j-k}) \\
 & & \quad \text{if } v^j \in M \wedge 0 < k \leq j \vee j \leq k < 0 \\
match_{[]}(P^x,M) & = & (match(P,v) \oplus \{x \mapsto k\},\ M - v^j + v^{j-k}) \\
 & & \quad \text{if } v^j \in M \wedge 0 < k \leq j \vee j \leq k < 0 \\
match_{[]}(P^{\overline{x}},M) & = & (match(P,v) \oplus \{x \mapsto j\},\ M - v^j) \\
 & & \quad \text{if } v^j \in M \\
match_{[]}(P,M) & = & \mathbf{fail} \\
 & & \quad \text{otherwise}
\end{array}
$$

Figure 3: Pattern-matching molecules in set representation.

tracting non deterministically a value v^k (k lying between 0 and j). Matching $P^{\overline{x}}$ extracts v^j and associates x with j.

Chemical reactions are rephrased in this setting as follows:

$$\langle(\gamma(P)\lfloor C\rfloor.M), N\rangle \rightarrow \langle[\phi M, Y]\rangle \qquad \text{if } match_{[]}(P, N) = (\phi, Y) \wedge \phi C$$

A reaction can be decomposed in three steps:

1. A molecule X matching P and satisfying the reaction condition is extracted from the solution N. Pattern-matching yields a substitution ϕ and the remainder Y such that $N = [X, Y]$.
2. The body of the reaction is produced (i.e. ϕ is applied and the expressions in ϕM are reduced).
3. The result of the reaction (ϕM) is put back in the solution Y. Since the reaction may produce atoms which are already present in the solution, $(\phi M, Y)$ must be normalized in set-representation using $[\,]$. In an implementation, this would boil down to updating multiplicities in the table representing the current solution. Note that the normalization may produce a dynamic error e.g. if $v^{-\infty}$ occurs in Y and ϕM contains v^{∞} (or vice versa).

For example,

$$\begin{array}{l}\textbf{let}\, prod = \textbf{replace}\, x, y\, \textbf{by}\, x * y\, \textbf{in}\\ \textbf{let}\, rmunit = \gamma(\langle x^{\overline{y}}, \omega\rangle)\lfloor x = 1\rfloor.\omega\, \textbf{in}\\ \langle prod, rmunit, \langle 1^4, 2, 3, 6\rangle\rangle\\ \rightarrow \langle prod, 2, 3, 6\rangle\\ \rightarrow \langle prod, 6^2\rangle\\ \rightarrow \langle prod, 36\rangle\end{array}$$

The reaction $rmunit$ extracts the sub-solution after having removed all the occurrences of 1's in one step. The n-shot reaction $prod$ computes $2 * 3$; then the solution is normalized to $\langle prod, 6^2\rangle$ (i.e. 6's are grouped). The last reaction (where x and y each matches an occurrence of 6) yields 36.

For simplicity reasons, we have formalized reactions on set-representations as a global operation. In practice, the first and last steps needs an atomic access only on the entries (atoms) they modify. The second step can be done in parallel with other reactions. A real implementation would extract only a (smartly chosen) selection of atoms for pattern-matching and would update only the entries (value, multiplicity) corresponding to the atoms produced by the reaction. Therefore, reactions involving different atoms could take place in parallel.

The framework presented in this section does not describe a complete implementation which would require other refinements. However, it does show a representation of solutions which allows to explicitly manipulate constant, infinite and negative multiplicities. A drawback is that we lose some locality in the reactions, but this is unavoidable with patterns matching all the occurrences of a specific value in a solution.

7 Related work and perspectives

To the best of our knowledge, Gamma (Banâtre & Le Métayer 1986, Banâtre & Le Métayer 1993) was the first chemical model to be proposed. It consists in a single multiset containing basic inactive

molecules and external reactions. Reactions are n-shot: they are applied until no reaction can take place. They are first-order: they are not part of the multiset and cannot be taken as argument or returned as result. Moreover, there is no nested solutions. Even if sub-solutions can be encoded, there is no notion of inertia in Gamma (only global termination). A standard Gamma program is easily expressed in the γ-calculus as a solution with a collection of recursive γ-abstractions representing the reactions and a sub-solution of values representing the multiset. Gamma has inspired many extensions (e.g. composition operators (Hankin, et al. 1992)) and other chemical models.

The chemical abstract machine (Berry & Boudol 1992) (CHAM) is a chemical approach introduced to describe concurrent computations without explicit control. It started from Gamma and added many features such as membranes, (sub)solutions, inertia and airlocks. Like Gamma, reactions are n-shot rewrite rules which are not part of the multisets. The selection pattern in the left-hand side of rewrite rules can include constants which is a form of reaction condition. For example, in (Berry & Boudol 1992), the description of the operational semantics of the TCCS and CCS calculi contains a cleanup rule ($0 \rightharpoonup$) which removes molecules equal to 0. The CHAM would be equivalent to the γ-calculus if it was higher-order.

Our minimal chemical calculus is quite close to Berry and Boudol's concurrent λ-calculus (referred to here as the γ_{lb}-calculus) introduced after the chemical abstract machine (CHAM) in (Berry & Boudol 1992). The γ_{lb}-calculus relies also on variables, abstractions, an associative and commutative application operator and solutions. However, to distinguish between the γ-abstraction and its argument, it adds the notion of positive ions (denoted M^+). The γ-abstractions are negative ions (denoted $x^- M$) which can react only with positive ions:

$$\beta\text{-reaction: } (x^- M), N^+ \rightarrow M[x := N]$$

In fact, no reaction can occur within a positive ion and so arguments are passed unchanged to abstractions. Furthermore, an additional reduction law, the *hatching rule*, extracts an inert molecule M from a solution $\langle M \rangle$:

$$\text{hatching: } \langle W \rangle \rightleftharpoons W \qquad \text{if } W \text{ is inert}$$

In the γ-calculus, these two notions are replaced by the strict γ-reduction. In particular, hatching can be written explicitly as

$$(\gamma\langle x \rangle.x), \langle M \rangle$$

which extracts M from its solution when it becomes inert. Even if the γ-calculus looks simpler than the γ_{lb}-calculus, it seems that they cannot be translated easily into each other (e.g. by a translation defined on the syntax rules).

A first higher-order extension of Gamma has been proposed in (Le Métayer 1994). The definition of Gamma involves two different kinds of terms: the program (set of rewrite rules) and multisets. The main extension of higher-order Gamma consists in unifying these two categories of expression into a single notion of configuration. A configuration contains a program and a list of named multisets. It is denoted by $[Prog,\ Var_1 = Multiset_1, \ldots, Var_n = Multiset_n]$. The program $Prog$ is a rewrite rule of the multisets (named Var_i) of the configuration. This model is an higher-order model because any configuration can handle other configurations through their program. It includes reaction conditions and n-ary rewrite rules. However, reactions are not first-class citizens since they are kept separate from multisets of data.

The hmm-calculus (Cohen & Muylaert-Filho 1996) (for *higher-order multiset machines*) is described as an extension of Gamma where reactions are one-shot and first-class citizens. An abstraction denoted by $\lambda\tilde{x}.M_1 \Leftarrow M_0$ describes a reaction rule: it takes several terms denoted by a tuple $\tilde{x}$, the term M_1 is the action and the term M_0 is the reaction condition. Like γ_{lb}, the hmm-calculus uses a call-by-name strategy. It needs an hatching rule to extract an inert molecule from its solution. Any reaction can occur within solutions and within abstractions. The hmm-calculus can be seen as a call-by-name version of the γ-calculus, or as an extension of the γ_{lb}-calculus with conditional and n-ary reactions.

P-systems (Păun 2000) are computing devices inspired from biology. It consists in nested membranes in which molecules react. Molecules can cross and move between membranes. A set of partially ordered rewrite rules is associated to each membrane. These rules describe possible reactions and communications between membranes of molecules. These features can be expressed in HOCL: a membrane is a solution, i.e. a multiset.

Our list of comparisons is not exhaustive and other models could have been considered. For example, we can mention work out about concurrent λ-calculus according to a chemical metaphor such as (Fontana & Buss 1994), or, for example, various models based on real chemistry as described in (Dittrich, et al. 2001).

To summarize the main contributions of this paper, we can emphasize (1) the use of a very general version of multisets with elements possessing various kinds of (finite or infinite) multiplicities and (2) the introduction of a higher order model of computation (HOCL) dealing with such multisets. Several programming examples illustrate the salient features of the language. In order to simplify the presentation, we limited multiplets to basic values. This restriction can be relaxed in several ways. Multiplets of pairs or solutions of values would not cause any other problems than efficiency if these structures include too many elements. Note also that the equality relation which prevented the use of multiplets of reactions is only needed for matching multiplets. The definition of finite multiplets (M^V), which can be seen as syntactic sugar for V occurrences of M, could apply to reactions and to any molecule. With these modest extensions the Jackpot! program (see Program 3) would be more elegantly expressed as

$$\langle wheel^3, choose^3, win \rangle$$

A current research direction concerns the use of HOCL as a coordination language for the description of GRID systems and applications. The basic challenge consists in showing that the chemical paradigm, represented by HOCL, allows a clean and elegant expression of features such as program mobility, load balancing, crash recovery, etc. Basically, the overall system is expressed as a "soup" (represented by a multiset) of resources such as processors, storage, communication links, etc. whose combinations are described by appropriate reaction rules.

References

J.-P. Banâtre, et al. (2001). 'Gamma and the Chemical Reaction Model: Fifteen Years After'. In *Multiset Processing*, vol. 2235 of *LNCS*, pp. 17–44. Springer-Verlag.

J.-P. Banâtre, et al. (2004). 'Chemical Specification of Autonomic Systems'. In *Proc. of the 13th Int. Conf. on Intelligent and Adaptive Systems and Software Engineering (IASSE'04)*.

J.-P. Banâtre, et al. (2005a). 'Higher-order Programming Style'. In *Proc. of the workshop on Unconventional Programming Paradigms (UPP'04)*, vol. 3566 of *LNCS*. Springer-Verlag.

J.-P. Banâtre, et al. (2005b). 'Principles of Chemical Programming'. In S. Abdennadher & C. Ringeissen (eds.), *Proceedings of the 5th International Workshop on Rule-Based Programming (RULE 2004)*, vol. 124 of *ENTCS*, pp. 133–147. Elsevier.

J.-P. Banâtre & D. Le Métayer (1986). 'A new computational model and its discipline of programming'. Tech. Rep. RR0566, INRIA.

J.-P. Banâtre & D. Le Métayer (1993). 'Programming by Multiset Transformation'. *Communications of the ACM (CACM)* **36**(1):98–111.

G. Berry & G. Boudol (1992). 'The Chemical Abstract Machine'. *Theoretical Computer Science* **96**:217–248.

W. Blizard (1990). 'Negative Membership'. *Notre Dame Journal of Formal Logic* **31**(3):346–368.

W. Blizard (1991). 'The development of multiset theory'. *Modern Logic* **1**:319 – 352.

C. Calude, et al. (eds.) (2001). *Multiset Processing, Mathematical, Computer Science and Molecular Computing Points of View*, Lecture Notes on Computer Science. Springer-Verlag.

M. Chaudron (1994). 'Schedules for Multiset Transformer Programs'. Tech. Rep. tr94-36, Rijksuniversiteit Leiden.

D. Cohen & J. Muylaert-Filho (1996). 'Introducing a Calculus for Higher-Order Multiset Programming'. In *Coordination Languages and Models*, vol. 1061 of *LNCS*, pp. 124–141.

N. Dershowitz & Z. Manna (1979). 'Proving Termination with Multiset Orderings'. *Communications of the ACM* **22**(8):465–476.

P. Dittrich, et al. (2001). 'Artificial Chemistries – A Review'. *Artificial Life* **7**(3):225–275.

W. Fontana & L. Buss (1994). 'The Arrival of the Fittest: Toward a Theory of Biological Organization'. *Bulletin of Mathematical Biology* **56**.

C. Hankin, et al. (1992). 'A Calculus of Gamma Programs'. In *Languages and Compilers for Parallel Computing, 5th International Workshop*, vol. 757 of *LNCS*, pp. 342–355. Springer-Verlag.

D. E. Knuth (1981). *The Art of Computer Programming*, vol. 2 Seminumerical Algorithms of *Addison-Wesley Series in Computer Science and Information Processing*. Addison-Wesley, 2nd edn.

D. Le Métayer (1994). 'Higher-order multiset programming'. In A. M. S. (AMS) (ed.), *Proc. of the DIMACS workshop on specifications of parallel algorithms*, vol. 18 of *Dimacs Series in Discrete Mathematics*.

D. Loeb (1992). 'Sets with a negative number of elements'. *Advances in Mathematics* **91**:64–74.

G. Păun (2000). 'Computing with Membranes'. *Journal of Computer and System Sciences* **61**(1):108–143.