3.8 The Chemical Abstract Machine (CHAM)

The *chemical abstract machine*, or the *CHAM*, is both a model of concurrency and a specific operational semantics style. The states of a CHAM are metaphorically regarded as chemical solutions formed with floating molecules. Molecules can interact with each other by means of reactions. A reaction can involve several molecules and can change them, delete them, and/or create new molecules. One of the most appealing aspects of the chemical abstract machine is that its reactions can take place concurrently, unrestricted by context. To facilitate local computation and to represent complex data-structures, molecules can be nested by encapsulating groups of molecules as sub-solutions. The chemical abstract machine was proposed as an alternative to SOS and its variants, including reduction semantics with evaluation contexts, in an attempt to circumvent their limitations, particularly their lack of support for true concurrency.

CHAM Syntax

The *basic molecules* of a CHAM are ordinary algebraic terms over a user-defined syntax. Several molecules wrapped within a membrane form a *solution*, which is also a molecule. The CHAM uses the symbols $\{\!|$ and $|\!\}$ to denote membranes. For example, $\{\!| m_1\ m_2\ \ldots\ m_k |\!\}$ is a solution formed with the molecules m_1, m_2, …, m_k. The order of molecules in a solution is irrelevant, so a solution can be regarded as a multi-set, or a bag of molecules wrapped within a membrane. Since solutions are themselves molecules, we can have arbitrarily nested molecules. This nesting mechanism is generic for all CHAMs and is given by the following algebraic CFG:

$$\begin{array}{rcl} \textit{Molecule} & ::= & \textit{Solution} \mid \textit{Molecule} \triangleleft \textit{Solution} \\ \textit{Solution} & ::= & \{\!| \textbf{Bag}\{\textit{Molecule}\} |\!\} \end{array}$$

The operator $\_ \triangleleft \_$ is called the *airlock* operator and will be discussed shortly (under general CHAM laws), after we discuss the CHAM rules. When defining a CHAM, one is only allowed to extend the syntax of molecules, which implicitly also extends the syntax that the solution terms can use. However, one is not allowed to explicitly extend the syntax of solutions. In other words, solutions can only be built using the generic syntax above, on top of user-defined syntactic extensions of molecules. Even though we do not formalize it here (and we are not aware of other formulations elsewhere either), it is understood that one can have multiple types of molecules in a CHAM.

Specific CHAM Rules

In addition to extending the syntax of molecules, a CHAM typically also defines a set of *rules*, each rule being a rule schemata but called a rule for simplicity. A CHAM rule has the form

$$m_1\ m_2\ \ldots\ m_k \to m'_1\ m'_2\ \ldots m'_l$$

where $m_1, m_2, \ldots, m_k$ and $m'_1, m'_2, \ldots, m'_l$ are not necessarily distinct molecules (since CHAM rules are schemata, these molecule terms may contain meta-variables). Molecules appearing in a rule are restricted to contain only subsolution terms which are either solution meta-variables or otherwise have the form $\{\!| m |\!\}$, where m is some molecule term. For example, a CHAM rule cannot contain subsolution terms of the form $\{\!| m\ s |\!\}$, $\{\!| m_1\ m_2 |\!\}$, or $\{\!| m_1\ m_2\ s |\!\}$, with m, m_1, m_2 molecule terms and s solution term, but it can contain ones of the form $\{\!| m |\!\}$, $\{\!| m_1 \triangleleft \{\!| m_2 |\!\} |\!\}$, $\{\!| m \triangleleft s |\!\}$, etc. This restriction is justified by chemical intuitions, namely that matching inside a solution is a rather complex operation which needs special handling (the airlock operator $\_ \triangleleft \_$ is used for this purpose). Note that CHAM rules are unconditional, that is, they have no premises.

General CHAM Laws

Any chemical abstract machine obeys the four laws below. Let CHAM be[8] a chemical abstract machine. Below we assume that mol, mol', mol_1, etc., are arbitrary concrete molecules of CHAM (i.e., no meta-variables) and that sol, sol', etc., are concrete solutions of it. If sol is the solution $\{|mol_1, mol_2, \ldots, mol_k|\}$ and sol' is the solution $\{|mol'_1, mol'_2, \ldots, mol'_l|\}$, then $sol \uplus sol'$ is the solution $\{|mol_1, mol_2, \ldots, mol_k, mol'_1, mol'_2, \ldots, mol'_l|\}$.

1. ***The Reaction Law.*** Given a CHAM rule

$$m_1\ m_2\ \ldots\ m_k \to m'_1\ m'_2\ \ldots m'_l \quad \in \quad \text{CHAM}$$

if $mol_1, mol_2, \ldots, mol_k$ and $mol'_1, mol'_2, \ldots, mol'_l$ are (concrete) instances of $m_1\ m_2\ \ldots\ m_k$ and of $m'_1\ m'_2\ \ldots m'_l$ by a common substitution, respectively, then

$$\text{CHAM} \vdash \{|mol_1\ mol_2\ \ldots\ mol_k|\} \to \{|mol'_1\ mol'_2\ \ldots mol'_l|\}$$

2. ***The Chemical Law.*** Reactions can be performed freely within any solution:

$$\frac{\text{CHAM} \vdash sol \to sol'}{\text{CHAM} \vdash sol \uplus sol'' \to sol' \uplus sol''}$$

3. ***The Membrane Law.*** A subsolution can evolve freely in any solution context $\{|cxt[\square]|\}$:

$$\frac{\text{CHAM} \vdash sol \to sol'}{\text{CHAM} \vdash \{|cxt[sol]|\} \to \{|cxt[sol']|\}}$$

4. ***The Airlock Law.***

$$\text{CHAM} \vdash \{|mol|\} \uplus sol \leftrightarrow \{|mol \triangleleft sol|\}$$

Note the unusual fact that $m_1\ m_2\ \ldots\ m_k \to m'_1\ m'_2\ \ldots m'_l$ being a rule in CHAM does not imply that CHAM $\vdash m_1\ m_2\ \ldots\ m_k \to m'_1\ m'_2\ \ldots m'_l$. Indeed, the CHAM rules are regarded as descriptors of changes that can take place in solutions and only in solutions, while CHAM sequents are incarnations of those otherwise purely abstract rules. What may be confusing is that the same applies also when k and l (the numbers of molecules in the left-hand and right-hand sides of the CHAM rule) happen to be 1 and m_1 and m'_1 happen to be solutions that contain no meta-variables. The two $\to$ arrows, namely the one in CHAM rules and the one in CHAM sequents, ought to be different symbols; however, we adhere to the conventional CHAM notation which uses the same symbol for both. Moreover, when the CHAM is clear from context, we also follow the conventional notation and drop it from sequents, that is, we write $sol \to sol'$ instead of CHAM $\vdash sol \to sol'$. While we admit that these conventions may sometimes be confusing, in that $sol \to sol'$ can be a rule or a sequent or even both, we hope that the context makes it clear which one is meant.

[8]To avoid inventing new names, it is common to use CHAM both as an abbreviation for "the chemical abstract machine" and as a name of an arbitrary but fixed chemical abstract machine.

The Reaction Law says that CHAM rules can only apply in solutions (wrapped by a membrane), and not arbitrarily wherever they match. The Chemical Law says that once a reaction take place in a certain solution, it can take place in any other larger solution. In other words, the fact that a solution has more molecules than required by the rule does not prohibit the rule from applying. The Reaction and the Chemical laws together say that CHAM rules can apply inside any solutions having some molecules that match the left-had side of the CHAM rule. An interesting case is when the left-hand-side term of the CHAM rule has only one molecule, i.e., when $k = 1$, because the CHAM rule is still allowed to only apply within a solution; it cannot apply in other places where the left-hand side happens to match.

The Membrane Law says that reactions can take place in any solution context. Indeed, $\{\!|cxt[sol]|\!\}$ says that the solution sol (which is wrapped in a membrane) appears somewhere, anywhere, inside a solution context $\{\!|cxt[\square]|\!\}$. Here cxt can be any bag-of-molecule context, and we write $cxt[\square]$ to highlight the fact that it is a context with a hole $\square$. By wrapping $cxt[\square]$ in a membrane we enforce a solution context. This rule also suggests that, at any given moment, the global term to rewrite using the CHAM rules should be a solution. Indeed, the CHAM rewriting process gets stuck as soon as the term becomes a proper molecule (not a solution), because the Membrane Law cannot apply.

The Airlock Law is reversible (i.e., it comprises two rewrite rules, one from left-to-right and one from right-to-left) and it allows to extract a molecule from a solution, putting the rest of the solution within a membrane. Using this law one can, for example, rewrite a solution $\{\!|mol_1\ mol_2\ \ldots, mol_k|\!\}$ into $\{\!|mol_1 \triangleleft \{\!|mol_2\ \ldots\ mol_k|\!\}|\!\}$. The advantage of doing so is that one can now match the molecule mol_1 within other rules. Indeed, recall that sub-solutions that appear in rules cannot specify any particular molecule term among the rest of the solution, unless the solution contains precisely that molecule. Since $mol_1 \triangleleft \{\!|mol_2\ \ldots\ mol_k|\!\}$ is a molecule, $\{\!|mol_1 \triangleleft \{\!|mol_2\ \ldots\ mol_k|\!\}|\!\}$ can match molecule terms of the form $\{\!|m \triangleleft s|\!\}$ appearing in CHAM rules, this way one effectively matching (and possibly modifying) the molecule mol_1 via the specific CHAM rules. The Airlock Law is the only means provided by the CHAM to extract or put molecules in a solution.

Note that the four laws above are not claimed to define the CHAM rewriting; they are only properties that the CHAM rewriting should satisfy. In particular, these laws do not capture the concurrency potential of the CHAM.

Definition 19. *The four laws above give us a proof system for CHAM sequents. As usual, if we write* CHAM $\vdash sol \to sol'$ *in isolation we mean that it is derivable. Also as usual, we let* $\to^*$ *denote the reflexive and transitive closure of* $\to$*, that is,* CHAM $\vdash sol \to^* sol'$ *if and only if* $sol = sol'$ *or there is some* sol'' *such that* CHAM $\vdash sol \to sol''$ *and* CHAM $\vdash sol'' \to^* sol'$*. Finally, we write* CHAM $\vdash sol \leftrightarrow sol'$ *as a shorthand for the two seqents* CHAM $\vdash sol \to sol'$ *and* CHAM $\vdash sol' \to sol$*, and we say that it is derivable if and only if the two sequents are derivable.*

None of the two sequents in Definition 19 captures the underlying concurrent computation of the CHAM. Indeed, CHAM $\vdash sol \to sol'$ says that one and only one reaction takes place somewhere in sol, while CHAM $\vdash sol \to^* sol'$ says that arbitrarily many steps take place, including ones which can be done concurrently but also ones which can only take place sequentially. Therefore, we can think of the four laws above, and implicitly of the sequents CHAM $\vdash sol \to sol'$ and CHAM $\vdash sol \to^* sol'$, as expressing the descriptive capability of the CHAM: what is possible and what is not possible to compute using the CHAM, and not how it operates. We argue that it is still suggestive to think of CHAM reactions as taking place concurrently whenever they do not involve the same molecules, even though this notion is not formalized here. We are actually not aware of any works except this book that formalize the CHAM concurrency: one can find a representation of CHAM rewriting into K rewriting in Section 9.6, which allows the CHAM to borrow K's concurrent rewriting.

A common source of misunderstanding the CHAM is to wrongly think of CHAM rules as ordinary rewriting rules modulo the associativity, commutativity and identity of the molecule grouping (inside a solution) operation. The major distinction between CHAM rules and such rewrite rules is that the former only apply within solutions (which are wrapped by membranes) no matter whether the rule contains one or more molecules in its left-hand or right-hand terms, while the latter apply anywhere they match. For example, supposing that one extends the syntax of molecules with the syntax of IMP in Section 3.1.1 and one adds a CHAM rule $m + 0 \to m$, then one can rewrite the solution $\{\!|(3+0)\ 7|\!\}$ to solution $\{\!|3\ 7|\!\}$, but one cannot rewrite the molecule $5 \,/\, (3+0)$ to molecule $5 \,/\, 3$ regardless of what context it is in, because $3+0$ is not in a solution. One cannot even rewrite the isolated (i.e., not in a solution context) term $3+0$ to 3 in CHAM, for the same reason.

Classification of CHAM Rules

The rules of a CHAM are typically partitioned into three intuitive categories, namely *heating*, *cooling* and *reaction* rules, though there are no formal requirements imposing a rule to be into one category or another. Moreover, the same laws discussed above apply the same way to all categories of rules, and the same restrictions preventing multiset matching apply to all of them.

- *Heating* rules, distinguished by using the relation symbol $\rightharpoonup$ instead of $\to$, are used to structurally rearrange the solution so that reactions can take place.
- *Cooling* rules, distinguished by using the relation symbol $\rightharpoondown$ instead of $\to$, are used after reactions take place to structurally rearrange the solution back into a convenient form, including to remove useless molecules or parts of them.
- *Reaction* rules, which capture the intended computational steps and use the conventional rewrite symbol $\to$, are used to evolve the solution in an irreversible way.

The heating and cooling rules can typically be paired, with each heating rule $l \rightharpoonup r$ having a symmetric cooling rule $l \leftharpoondown r$, so that we can view them as a single bidirectional *heating/cooling* rule. The CHAM notation for writing such heating/cooling rules is the following:

$$l \rightleftharpoons r$$

In particular, it makes sense to regard the airlock axiom as an example of such a heating/cooling bidirectional rule, that is,

$$\{\!|m_1\ m_2\ \ldots\ m_k|\!\} \rightleftharpoons \{\!|m_1 \triangleleft \{\!|m_2\ \ldots\ m_k|\!\}|\!\}$$

where m_1, m_2, $\ldots$, m_k are molecule meta-variables. The intuition here is that we can heat the solution until m_1 is extracted in an airlock, or we can cool it down in which case the airlock m_1 is diffused within the solution. However, we need to assume one such rule for each $k > 0$.

As one may expect, the reaction rules are the heart of the Cham and properly correspond to state transitions. The heating and cooling rules express *structural rearrangements*, so that the reaction rules can match and apply. In other words, we can view the reaction rules as being applied *modulo* the heating and cooling rules. We are going to suggestively use the notation

CHAM $\vdash sol \rightharpoonup sol'$, respectively CHAM $\vdash sol \rightharpoondown sol'$ whenever the rewrite step taking sol to sol' is a heating rule, respectively a cooling rule. Similarly, we may use the notations CHAM $\vdash sol \rightharpoonup^* sol'$ and CHAM $\vdash sol \rightharpoondown^* sol'$ for the corresponding reflexive/transitive closures. Also, to emphasize the fact that there is only one reaction rule applied, we take the freedom to (admittedly ambiguously) write CHAM $\vdash sol \to sol'$ instead of CHAM $\vdash sol(\to \cup \rightharpoonup \cup \rightharpoondown)^* sol'$ whenever all the involved rules but one are heating or cooling rules.

3.8.1 The CHAM of IMP

We next show how to give IMP a CHAM semantics. CHAM is particularly well-suited to giving semantics to concurrent distributed calculi and languages, yielding considerably simpler definitions than those afforded by SOS. Since IMP is sequential, it cannot take full advantage of the CHAM's true concurrency capabilities; the multi-threaded IMP++ language discussed in Section 3.5 will make slightly better use of CHAM's capabilities. Nevertheless, some of CHAM's capabilities turn out to be useful even in this sequential language application, others turn out to be deceiving. Our CHAM semantics for IMP below follows in principle the reduction semantics with evaluation contexts definition discussed in Section 3.7.1. One can formally show that a step performed using reduction under evaluation contexts is equivalent to a suite of heating steps, followed by one reaction step, followed by a suite of cooling steps.

The CHAM defined below is just one possible way to give IMP a CHAM semantics. CHAM, like rewriting, is a general framework which does not impose upon its users any particular definitional style. In our case, we chose to conceptually distinguish two types of molecules; we say "conceptually" because, for simplicity, we prefer to define only one *Molecule* syntactic category in our CHAM:

- *Syntactic molecules*, which include all the syntax of IMP in Section 3.1.1, plus all the syntax of its evaluation contexts in Section 3.7.1, plus a mechanism to flatten evaluation contexts; again, for simplicity, we prefer not to include a distinct type of molecule for each distinct syntactic category of IMP.

- *State molecules*, which are pairs $x \mapsto i$, where $x \in Id$ and $i \in Int$.

For clarity, we prefer to keep the syntactic and the state molecules in separate solutions. More precisely, we work with top-level configurations which are solutions of the form

$$\{\!|\, \{\!|\, Syntax \,|\!\}\ \ \{\!|\, State \,|\!\} \,|\!\}$$

Syntax and *State* are solutions containing syntactic and state molecules, respectively. For example,

$$\{\!|\, \{\!|\, x := (3 \,/\, (x + 2)) \,|\!\}\ \{\!|\, x \mapsto 1\ \ y \mapsto 0 \,|\!\} \,|\!\}$$

is a CHAM configuration containing the statement $x := (3 \,/\, (x + 2))$ and state $x \mapsto 1, y \mapsto 0$.

The state molecules and implicitly the state solution are straightforward. State molecules are not nested and state solutions are simply multisets of molecules of the form $x \mapsto i$. The CHAM does not allow us to impose constraints on solutions, such as that the molecules inside the state solution indeed define a partial function and not some arbitrary relation (e.g., that there is at most one molecule $x \mapsto i$ for each $x \in Id$). Instead, the state solution will be used in such a way that the original state solution will embed a proper partial function and each rule will preserve this property of it. For example, the CHAM rule for variable assignment, say when assigning integer i to variable x, will match the state molecule as $\{\!|\, x \mapsto j \triangleleft \sigma \,|\!\}$ and will rewrite it to $\{\!|\, x \mapsto i \triangleleft \sigma \,|\!\}$.

The top level syntactic solution holds the current program or fragment of program that is still left to be processed. It is not immediately clear how the syntactic solution should be represented in order to be able to give IMP a CHAM semantics. The challenge here is that the IMP language constructs have evaluation strategies and the subterms that need to be next processed can be arbitrarily deep into the program or fragment of program, such as the framed x in $x := (3 / (\boxed{x} + 2))$. If the CHAM allowed conditional rules, then we could have followed an approach similar to that of SOS described in Section 3.3, reducing the semantics of each language construct to that of its subexpressions or substatements. Similarly, if the CHAM allowed matching using evaluation contexts, then we could have followed a reduction semantics with evaluation contexts approach like the one in Section 3.7.1 which uses only unconditional rules. Unfortunately, the CHAM allows neither conditional rules nor evaluation contexts in matching, so a different approach is needed.

Failed attempts to represent syntax. A natural approach to represent the syntax of a programming language in CHAM may be to try to use CHAM's heating/cooling and molecule/solution nesting mechanisms to decompose syntax unambiguously in such a way that the redex (i.e., the subterm which can be potentially reduced next; see Section 3.7) appears as a molecule in the top syntactic solution. That is, if $p = c[t]$ is a program or fragment of program which can be decomposed in evaluation context c and redex t, then one may attempt to represent it as a solution of the form $\{\!| t\ \gamma_c |\!\}$, where γ_c is some CHAM representation of the evaluation context c. If this worked, then we could use an airlock operation to isolate that redex from the rest of the syntactic solution, i.e. $\{\!| p |\!\} \rightleftharpoons \{\!| t\ \gamma_c |\!\} \rightleftharpoons \{\!| t \triangleleft \{\!| \gamma_c |\!\} |\!\}$, and thus have it at the same level with the state solution in the configuration solution; this would allow to have rules that match both a syntactic molecule and a state molecule (after an airlock operation is applied on the state solution as well) in the same rule, as needed for the semantics of lookup and assignment. In our example above, we would obtain

$$\{\!| \{\!| x := (3 / (x + 2)) |\!\}\ \{\!| x \mapsto 1\quad y \mapsto 0 |\!\} |\!\} \rightleftharpoons \{\!| \{\!| x \triangleleft \{\!| \gamma_{x := (3 / (\square + 2))} |\!\} |\!\}\ \{\!| x \mapsto 1 \triangleleft \{\!| y \mapsto 0 |\!\} |\!\} |\!\}$$

and the latter could be rewritten with a natural CHAM reaction rule for variable lookup such as

$$\{\!| x \triangleleft c |\!\}\ \{\!| x \mapsto i \triangleleft \sigma |\!\} \to \{\!| i \triangleleft c |\!\}\ \{\!| x \mapsto i \triangleleft \sigma |\!\}$$

Unfortunately, there seems to be no way to achieve such a desirable CHAM representation of syntax. We next attempt and fail to do it in two different ways, and then give an argument why such a representation is actually impossible.

Consider, again, the statement $x := (3 / (x + 2))$. A naive approach to represent this statement term as a syntactic solution (by means of appropriate heating/cooling rules) is to flatten it into its redex, namely x, and into all its atomic evaluation subcontexts, that is, to represent it as the following solution:

$$\{\!| x\quad (\square + 2)\quad (3 / \square)\quad (x := \square) |\!\}$$

Such a representation can be relatively easily achieved by adding heating/cooling pair rules that correspond to the evaluation strategies (or contexts) of the various language constructs. For example, we can add the following rules corresponding to the evaluation strategies of the assignment and the addition constructs (and two similar ones for the division construct):

$$\begin{array}{rcl} x := a & \rightleftharpoons & a \triangleleft \{\!| x := \square |\!\} \\ a_1 + a_2 & \rightleftharpoons & a_1 \triangleleft \{\!| \square + a_2 |\!\} \\ a_1 + a_2 & \rightleftharpoons & a_2 \triangleleft \{\!| a_1 + \square |\!\} \end{array}$$

With such rules, one can now heat or cool syntax as desired, for example:

$$\begin{array}{rcll}
\{\!|x := (3\,/\,(x+2))|\!\} & \rightleftharpoons & \{\!|(3\,/\,(x+2)) \lhd \{\!|x := \square\,|\!\}|\!\} & \text{(Reaction)}\\
& \rightleftharpoons & \{\!|(3\,/\,(x+2))\ \ (x := \square)|\!\} & \text{(Airlock)}\\
& \rightleftharpoons & \{\!|(x+2)\ \ (3\,/\,\square)\ \ (x := \square)|\!\} & \text{(Reaction, Chemical, Airlock)}\\
& \rightleftharpoons & \{\!|x\ \ (\square+2)\ \ (3\,/\,\square)\ \ (x := \square)|\!\} & \text{(Reaction, Chemical, Airlock)}
\end{array}$$

Unfortunately this naive approach is ambiguous, because it cannot distinguish the above from the representation of, say, $x := ((3\,/\,x)+2)$. The problem here is that the precise structure of the evaluation context is "lost in translation", so the approach above does not work.

Let us attempt a second approach, namely to guarantee that there is precisely one hole $\square$ molecule in each syntactic subsolution by using the molecule/solution nesting mechanism available in CHAM. More precisely, let us try to unambiguously represent the statements $x := (3\,/\,(x+2))$ and $x := ((3\,/\,x)+2)$ as the following two distinct syntactic solutions:

$$\begin{array}{c}
\{\!|x\ \ \{\!|(\square+2)\ \ \{\!|(3\,/\,\square)\ \ \{\!|(x := \square)|\!\}|\!\}|\!\}|\!\}\\
\{\!|x\ \ \{\!|(3\,/\,\square)\ \ \{\!|(\square+2)\ \ \{\!|(x := \square)|\!\}|\!\}|\!\}|\!\}
\end{array}$$

To achieve this, we modify the heating/cooling rules above as follows:

$$\begin{array}{rcl}
(x := a) \lhd c & \rightleftharpoons & a \lhd \{\!|\{\!|(x := \square) \lhd c|\!\}|\!\}\\
(a_1 + a_2) \lhd c & \rightleftharpoons & a_1 \lhd \{\!|\{\!|(\square + a_2) \lhd c|\!\}|\!\}\\
(a_1 + a_2) \lhd c & \rightleftharpoons & a_2 \lhd \{\!|\{\!|(a_1 + \square) \lhd c|\!\}|\!\}
\end{array}$$

With these modified rules, one may now think that one can heat and cool syntax unambiguously:

$$\begin{array}{rcll}
\{\!|x := (3\,/\,(x+2))|\!\} & \rightleftharpoons & \{\!|(x := (3\,/\,(x+2))) \lhd \{\!|\cdot|\!\}|\!\} & \text{(Airlock)}\\
& \rightleftharpoons & \{\!|(3\,/\,(x+2)) \lhd \{\!|\{\!|(x := \square) \lhd \{\!|\cdot|\!\}|\!\}|\!\}|\!\} & \text{(Reaction)}\\
& \rightleftharpoons & \{\!|(3\,/\,(x+2)) \lhd \{\!|\{\!|x := \square\,|\!\}|\!\}|\!\} & \text{(Airlock, Membrane)}\\
& \rightleftharpoons & \{\!|(x+2) \lhd \{\!|\{\!|(3\,/\,\square) \lhd \{\!|\{\!|x := \square\,|\!\}|\!\}|\!\}|\!\}|\!\} & \text{(Reaction)}\\
& \rightleftharpoons & \{\!|(x+2) \lhd \{\!|\{\!|(3\,/\,\square)\ \ \{\!|x := \square\,|\!\}|\!\}|\!\}|\!\} & \text{(Airlock, Membrane)}\\
& \rightleftharpoons & \{\!|x \lhd \{\!|\{\!|(\square+2) \lhd \{\!|\{\!|(3\,/\,\square)\ \ \{\!|x := \square\,|\!\}|\!\}|\!\}|\!\}|\!\}|\!\} & \text{(Reaction)}\\
& \rightleftharpoons & \{\!|x \lhd \{\!|\{\!|(\square+2)\ \ \{\!|(3\,/\,\square)\ \ \{\!|x := \square\,|\!\}|\!\}|\!\}|\!\}|\!\} & \text{(Airlock, Membrane)}\\
& \rightleftharpoons & \{\!|x\ \ \{\!|(\square+2)\ \ \{\!|(3\,/\,\square)\ \ \{\!|x := \square\,|\!\}|\!\}|\!\}|\!\} & \text{(Airlock)}
\end{array}$$

Unfortunately, the above is not the only way one can heat the solution in question. For example, the following is also a possible derivation, showing that these heating/cooling rules are still problematic:

$$\begin{array}{rcll}
\{\!|x := (3\,/\,(x+2))|\!\} & \rightleftharpoons & \{\!|(x := (3\,/\,(x+2))) \lhd \{\!|\cdot|\!\}|\!\} & \text{(Airlock)}\\
& \rightleftharpoons & \{\!|(3\,/\,(x+2)) \lhd \{\!|\{\!|(x := \square) \lhd \{\!|\cdot|\!\}|\!\}|\!\}|\!\} & \text{(Reaction)}\\
& \rightleftharpoons & \{\!|(3\,/\,(x+2)) \lhd \{\!|\{\!|x := \square\,|\!\}|\!\}|\!\} & \text{(Airlock, Membrane)}\\
& \rightleftharpoons & \{\!|(3\,/\,(x+2))\ \ \{\!|x := \square\,|\!\}|\!\} & \text{(Airlock)}\\
& \rightleftharpoons & \{\!|(x+2)\ \ \{\!|3\,/\,\square\,|\!\}\ \ \{\!|x := \square\,|\!\}|\!\} & \text{(All four laws)}\\
& \rightleftharpoons & \{\!|x\ \ \{\!|\,\square+2|\!\}\ \ \{\!|3\,/\,\square\,|\!\}\ \ \{\!|x := \square\,|\!\}|\!\} & \text{(All four laws)}
\end{array}$$

Indeed, one can similarly show that

$$\{\!|x := ((3\,/\,x)+2)|\!\} \quad \rightleftharpoons \quad \{\!|x\ \ \{\!|\,\square+2|\!\}\ \ \{\!|3\,/\,\square\,|\!\}\ \ \{\!|x := \square\,|\!\}|\!\}$$

Therefore, this second syntax representation attempt is also ambiguous.

$$\begin{array}{rcl}
a_1 \,\texttt{+}\, a_2 \curvearrowright c & \rightleftharpoons & a_1 \curvearrowright \square \,\texttt{+}\, a_2 \curvearrowright c \\
a_1 \,\texttt{+}\, a_2 \curvearrowright c & \rightleftharpoons & a_2 \curvearrowright a_1 \,\texttt{+}\, \square \curvearrowright c \\
a_1 \,\texttt{/}\, a_2 \curvearrowright c & \rightleftharpoons & a_1 \curvearrowright \square \,\texttt{/}\, a_2 \curvearrowright c \\
a_1 \,\texttt{/}\, a_2 \curvearrowright c & \rightleftharpoons & a_2 \curvearrowright a_1 \,\texttt{/}\, \square \curvearrowright c \\
a_1 \,\texttt{<=}\, a_2 \curvearrowright c & \rightleftharpoons & a_1 \curvearrowright \square \,\texttt{<=}\, a_2 \curvearrowright c \\
i_1 \,\texttt{<=}\, a_2 \curvearrowright c & \rightleftharpoons & a_2 \curvearrowright i_1 \,\texttt{<=}\, \square \curvearrowright c \\
\texttt{not}\, b \curvearrowright c & \rightleftharpoons & b \curvearrowright \texttt{not}\, \square \curvearrowright c \\
b_1 \,\texttt{and}\, b_2 \curvearrowright c & \rightleftharpoons & b_1 \curvearrowright \square \,\texttt{and}\, b_2 \curvearrowright c \\
x \,\texttt{:=}\, a \curvearrowright c & \rightleftharpoons & a \curvearrowright x \,\texttt{:=}\, \square \curvearrowright c \\
s_1 \,\texttt{;}\, s_2 \curvearrowright c & \rightleftharpoons & s_1 \curvearrowright \square \,\texttt{;}\, s_2 \curvearrowright c \\
s & \rightleftharpoons & s \curvearrowright \square \\
\texttt{if}\, b \,\texttt{then}\, s_1 \,\texttt{else}\, s_2 \curvearrowright c & \rightleftharpoons & b \curvearrowright \texttt{if}\ \square\ \texttt{then}\, s_1 \,\texttt{else}\, s_2 \curvearrowright c
\end{array}$$

Figure 3.44: CHAM heating-cooling rules for IMP.

We claim that it is impossible to devise heating/cooling rules in CHAM and representations $\gamma_{\_}$ of evaluation contexts with the property that

$$\{\!|c[t]|\!\} \rightleftharpoons \{\!|t\ \gamma_c|\!\} \quad \text{or, equivalently,} \quad \{\!|c[t]|\!\} \rightleftharpoons \{\!|t \triangleleft \{\!|\gamma_c|\!\}|\!\}$$

for any term t and any appropriate evaluation context c. Indeed, if that was possible, then the following derivation could be possible:

$$\begin{array}{rcll}
\{\!|x \,\texttt{:=}\, (3 \,\texttt{/}\, (x \,\texttt{+}\, 2))|\!\} & \rightleftharpoons & \{\!|(3 \,\texttt{/}\, (x \,\texttt{+}\, 2))\ \ \gamma_{x \,\texttt{:=}\, \square}|\!\} & \text{(hypothesis)} \\
& \rightleftharpoons & \{\!|(x \,\texttt{+}\, 2)\ \ \gamma_{3 \,\texttt{/}\, \square}\ \ \gamma_{x \,\texttt{:=}\, \square}|\!\} & \text{(hypothesis, Chemical)} \\
& \rightleftharpoons & \{\!|x\ \ \gamma_{\square \,\texttt{+}\, 2}\ \ \gamma_{3 \,\texttt{/}\, \square}\ \ \gamma_{x \,\texttt{:=}\, \square}|\!\} & \text{(hypothesis, Chemical)} \\
& \rightleftharpoons & \{\!|(3 \,\texttt{/}\, x)\ \ \gamma_{\square \,\texttt{+}\, 2}\ \ \gamma_{x \,\texttt{:=}\, \square}|\!\} & \text{(hypothesis, Chemical)} \\
& \rightleftharpoons & \{\!|((3 \,\texttt{/}\, x) \,\texttt{+}\, 2)\ \ \gamma_{x \,\texttt{:=}\, \square}|\!\} & \text{(hypothesis, Chemical)} \\
& \rightleftharpoons & \{\!|x \,\texttt{:=}\, ((3 \,\texttt{/}\, x) \,\texttt{+}\, 2)|\!\} & \text{(hypothesis)}
\end{array}$$

This general impossibility result explains why both our representation attempts above failed, as well as why many other similar attempts are also expected to fail.

The morale of the exercise above is that one should be very careful when using CHAM's airlock, because in combination with the other CHAM laws it can yield unexpected behaviors. In particular, the Chemical Law makes it impossible to state that a term matches the entire contents of a solution molecule, so one should not rely on the fact that all the remaining contents of a solution is in the membrane following the airlock. In our heating/cooling rules above, for example

$$\begin{array}{rcl}
(x \,\texttt{:=}\, a) \triangleleft c & \rightleftharpoons & a \triangleleft \{\!|\{\!|(x \,\texttt{:=}\, \square) \triangleleft c|\!\}|\!\} \\
(a_1 \,\texttt{+}\, a_2) \triangleleft c & \rightleftharpoons & a_1 \triangleleft \{\!|\{\!|(\square \,\texttt{+}\, a_2) \triangleleft c|\!\}|\!\} \\
(a_1 \,\texttt{+}\, a_2) \triangleleft c & \rightleftharpoons & a_2 \triangleleft \{\!|\{\!|(a_1 \,\texttt{+}\, \square) \triangleleft c|\!\}|\!\}
\end{array}$$

our intuition that c matches *all* the evaluation context solution representation was wrong precisely for that reason. Indeed, it can just as well match a solution representation of a subcontext, which is why we got the unexpected derivation.

Correct representation of syntax. We next discuss an approach to representing syntax which is *not* based on CHAM's existing solution/membrane mechanism. We borrow from K (see Chapter 5) the idea of flattening syntax in an explicit list of computational tasks. Like in K, we use the symbol $\curvearrowright$, read "then" or "followed by", to separate such computational tasks; to avoid writing parentheses, we here assume that $\curvearrowright$ is right-associative and binds less tightly than any other construct. For example, the term $x := (3 \,/\, (x + 2))$ gets represented as the list term

$$x \curvearrowright \square + 2 \curvearrowright 3 \,/\, \square \curvearrowright x := \square \curvearrowright \square$$

which reads "process x, followed by adding 2 to it, followed by dividing 3 by the result, followed by assigning the obtained result to x, which is the final task". Figure 3.44 shows all the heating/cooling rules that we associate to the various evaluation strategies of the IMP language constructs. These rules allow us to structurally rearrange any well-formed syntactic term so that the next computational task is at the top (left side) of the computation list. The only rule in Figure 3.44 which does not correspond to the evaluation strategy of some evaluation construct is $s \rightleftharpoons s \curvearrowright \square$. Its role is to initiate the decomposition process whenever an unheated statement is detected in the syntax solution. According to CHAM's laws, these rules can only apply in solutions, so we can derive

$$\{\!|x := 1 \,;\, x := (3 \,/\, (x + 2))|\!\} \rightleftharpoons^* \{\!|x := 1 \curvearrowright \square \,;\, x := (3 \,/\, (x + 2)) \curvearrowright \square|\!\}$$

but there is no way to derive, for example,

$$\{\!|x := 1 \,;\, x := (3 \,/\, (x + 2))|\!\} \rightleftharpoons^* \{\!|x := 1 \,;\, (x \curvearrowright \square + 2 \curvearrowright 3 \,/\, \square \curvearrowright x := \square \curvearrowright \square)|\!\}$$

The syntactic solution will contain only one syntactic molecule at any given moment, with no subsolutions, which is the reason why the heating/cooling rules in Figure 3.44 that correspond to language construct evaluation strategies need to mention the remaining of the list of computational tasks in the syntactic molecule, c, instead of just the interesting part (e.g., $a_1 + a_2 \rightleftharpoons a_1 \curvearrowright \square + a_2$, etc.), as we do in K (see Section 5). The heating/cooling rules in Figure 3.44 effectively decompose the syntactic term into any of its possible splits into a redex (the top of the resulting list of computational tasks) and an evaluation context (represented flattened as the rest of the list). In fact, these heating/cooling rules have been almost mechanically derived from the syntax of evaluation contexts for the IMP language constructs in Section 3.7.1 (see Figure 3.30).

Figure 3.45 shows the remaining CHAM rules of IMP, giving the actual semantics of each language construct. Like the heating/cooling rules in Figure 3.44, these rules are also almost mechanically derived from the rules of the reduction semantics with evaluation contexts of IMP in Section 3.7.1 (see Figure 3.31), with the following notable differences:

- Each rule needs to mention the remaining list of computational tasks, c, for the same reason the heating/cooling rules in Figure 3.44 need to mention it (which is explained above).
- There is no equivalent of the characteristic rule of reduction semantics with evaluation contexts. The Membrane Law looks somehow similar, but we cannot take advantage of that because we were not able to use the inherent airlock mechanism of the CHAM to represent syntax (see the failed attempts to represent syntax above).
- The state is organized as a solution using CHAM's airlock mechanism, instead of just imported as an external data-structure as we did in our previous semantics. We did so because the state

$$
\begin{array}{rcl}
\{\!|x \curvearrowright c|\!\}\ \{\!|x \mapsto i \rhd \sigma|\!\} & \to & \{\!|i \curvearrowright c|\!\}\ \{\!|x \mapsto i \rhd \sigma|\!\} \\
i_1 \,\texttt{+}\, i_2 \curvearrowright c & \to & i_1 +_{Int} i_2 \curvearrowright c \\
i_1 \,\texttt{/}\, i_2 \curvearrowright c & \to & i_1 /_{Int} i_2 \curvearrowright c \quad \text{when } i_2 \neq 0 \\
i_1 \,\texttt{<=}\, i_2 \curvearrowright c & \to & i_1 \leq_{Int} i_2 \curvearrowright c \\
\texttt{not true} \curvearrowright c & \to & \texttt{false} \curvearrowright c \\
\texttt{not false} \curvearrowright c & \to & \texttt{true} \curvearrowright c \\
\texttt{true and}\, b_2 \curvearrowright c & \to & b_2 \curvearrowright c \\
\texttt{false and}\, b_2 \curvearrowright c & \to & \texttt{false} \curvearrowright c \\
\{\!|x \,\texttt{:=}\, i \curvearrowright c|\!\}\ \{\!|x \mapsto j \rhd \sigma|\!\} & \to & \{\!|\texttt{skip} \curvearrowright c|\!\}\ \{\!|x \mapsto i \rhd \sigma|\!\} \\
\texttt{skip ;}\, s_2 \curvearrowright c & \to & s_2 \curvearrowright c \\
\texttt{if true then}\, s_1\, \texttt{else}\, s_2 \curvearrowright c & \to & s_1 \curvearrowright c \\
\texttt{if false then}\, s_1\, \texttt{else}\, s_2 \curvearrowright c & \to & s_2 \curvearrowright c \\
\texttt{while}\, b\, \texttt{do}\, s \curvearrowright c & \to & \texttt{if}\, b\, \texttt{then}\, (s\, \texttt{;}\, \texttt{while}\, b\, \texttt{do}\, s)\, \texttt{else skip} \curvearrowright c \\
\texttt{var}\, xl\, \texttt{;}\, s & \to & \{\!|s|\!\}\ \{\!|xl \mapsto 0|\!\} \\
\\
(x, xl) \mapsto i & \rightharpoonup & x \mapsto i \rhd \{\!|xl \mapsto i|\!\}
\end{array}
$$

Figure 3.45: CHAM(IMP): The CHAM of IMP, obtained by adding to the heating/cooling rules in Figure 3.44 the semantic rules for IMP plus the heating rule for state initialization above.

data-structure that we used in our previous semantics was a finite-domain partial function (see Section 3.1.2), which was represented as a set of pairs (Section 2.1.2), and it is quite natural to replace any set structures by the inherent CHAM solution mechanism.

Figure 3.46 shows a possible execution of IMP's CHAM defined above, mentioning at each step which of CHAM's laws have been applied. Note that the final configuration contains two subsolutions, one for the syntax and one for the state. Since the syntactic subsolution in the final configuration solution is expected to always contain only `skip`, one can safely eliminated it.

3.8.2 The CHAM of IMP++

We next discuss the CHAM of IMP++, discussing like in the other semantics each feature separately first and then putting all of them together. When putting them together, we also investigate the modularity and appropriateness of the resulting definition.

Variable Increment

The chemical abstract machine can also define the increment modularly:

$$\{\!|\texttt{++}x \curvearrowright c|\!\}\ \{\!|x \mapsto i \rhd \sigma|\!\} \to \{\!|i +_{Int} 1 \curvearrowright c|\!\}\ \{\!|x \mapsto i +_{Int} 1 \rhd \sigma|\!\} \qquad \text{(CHAM-Inc)}$$

Input/Output

All we have to do is to add new molecules in the top-level solution that hold the input and the output buffers, then define the evaluation strategy of `print` by means of a heating/cooling pair like we did for other strict constructs, and finally to add the reaction rules corresponding to the input output constructs. Since solutions are not typed, to distinguish the solution holding the input

$$
\begin{array}{rll}
\{\!|\,\texttt{var}\,x,y\,;\,x:=1\,;\,x:=(3\,/\,(x+2))\,|\!\} & \rightarrow & \text{(Reaction)}\\
\{\!|\{\!|x:=1\,;\,x:=(3\,/\,(x+2))|\!\}\ \ \{\!|x,y\mapsto 0|\!\}|\!\} & \rightharpoonup & \text{(Heating, Membrane)}\\
\{\!|\{\!|x:=1\,;\,x:=(3\,/\,(x+2))\curvearrowright\square|\!\}\ \ \{\!|x,y\mapsto 0|\!\}|\!\} & \rightharpoonup^{*} & \text{(Heating, Membrane, Airlock)}\\
\{\!|\{\!|x:=1\,;\,x:=(3\,/\,(x+2))\curvearrowright\square|\!\}\ \ \{\!|x\mapsto 0\ \ y\mapsto 0|\!\}|\!\} & \rightharpoonup & \text{(Heating, Membrane)}\\
\{\!|\{\!|x:=1\curvearrowright\square\,;\,x:=(3\,/\,(x+2))\curvearrowright\square|\!\}\ \ \{\!|x\mapsto 0\ \ y\mapsto 0|\!\}|\!\} & \rightharpoonup & \text{(Airlock, Membrane)}\\
\{\!|\{\!|x:=1\curvearrowright\square\,;\,x:=(3\,/\,(x+2))\curvearrowright\square|\!\}\ \ \{\!|x\mapsto 0\rhd\{\!|y\mapsto 0|\!\}|\!\}|\!\} & \rightarrow & \text{(Reaction)}\\
\{\!|\{\!|\,\texttt{skip}\curvearrowright\square\,;\,x:=(3\,/\,(x+2))\curvearrowright\square|\!\}\ \ \{\!|x\mapsto 1\rhd\{\!|y\mapsto 0|\!\}|\!\}|\!\} & \rightharpoondown & \text{(Cooling, Membrane)}\\
\{\!|\{\!|\,\texttt{skip}\,;\,x:=(3\,/\,(x+2))\curvearrowright\square|\!\}\ \ \{\!|x\mapsto 1\rhd\{\!|y\mapsto 0|\!\}|\!\}|\!\} & \rightharpoondown & \text{(Reaction, Membrane)}\\
\{\!|\{\!|x:=(3\,/\,(x+2))\curvearrowright\square|\!\}\ \ \{\!|x\mapsto 1\rhd\{\!|y\mapsto 0|\!\}|\!\}|\!\} & \rightharpoonup^{*} & \text{(Heating, Membrane)}\\
\{\!|\{\!|x\curvearrowright\square+2\curvearrowright 3\,/\,\square\curvearrowright x:=\square\curvearrowright\square|\!\}\ \ \{\!|x\mapsto 1\rhd\{\!|y\mapsto 0|\!\}|\!\}|\!\} & \rightarrow & \text{(Reaction)}\\
\{\!|\{\!|1\curvearrowright\square+2\curvearrowright 3\,/\,\square\curvearrowright x:=\square\curvearrowright\square|\!\}\ \ \{\!|x\mapsto 1\rhd\{\!|y\mapsto 0|\!\}|\!\}|\!\} & \rightharpoondown & \text{(Cooling)}\\
\{\!|\{\!|1+2\curvearrowright 3\,/\,\square\curvearrowright x:=\square\curvearrowright\square|\!\}\ \ \{\!|x\mapsto 1\rhd\{\!|y\mapsto 0|\!\}|\!\}|\!\} & \rightarrow & \text{(Reaction, Membrane)}\\
\{\!|\{\!|3\curvearrowright 3\,/\,\square\curvearrowright x:=\square\curvearrowright\square|\!\}\ \ \{\!|x\mapsto 1\rhd\{\!|y\mapsto 0|\!\}|\!\}|\!\} & \rightarrow^{*} & \text{(Reaction, Cooling, Membrane)}\\
\{\!|\{\!|x:=1\curvearrowright\square|\!\}\ \ \{\!|x\mapsto 1\rhd\{\!|y\mapsto 0|\!\}|\!\}|\!\} & \rightarrow & \text{(Reaction)}\\
\{\!|\{\!|\,\texttt{skip}\curvearrowright\square|\!\}\ \ \{\!|x\mapsto 1\rhd\{\!|y\mapsto 0|\!\}|\!\}|\!\} & \rightarrow & \text{(Cooling, Membrane)}\\
\{\!|\{\!|\,\texttt{skip}\,|\!\}\ \ \{\!|x\mapsto 1\rhd\{\!|y\mapsto 0|\!\}|\!\}|\!\} & &
\end{array}
$$

Figure 3.46: An execution of IMP's CHAM.

buffer from the one holding the output buffer, we introduce two artificial molecules, called input and output, respectively, and place them upfront in their corresponding solutions. Since the input buffer needs to also be provided within the initial solution, we modify the reaction rule for programs to take program and input molecules and initialize the output and state molecule accordingly:

$$
\begin{array}{ll}
\texttt{print(}\,a\,\texttt{)}\curvearrowright c \rightleftharpoons a\curvearrowright\texttt{print(}\ \square\ \texttt{)}\curvearrowright c & \\
\{\!|\texttt{read()}\curvearrowright c|\!\}\ \{\!|\mathsf{input}\ i:w|\!\}\rightarrow\{\!|\texttt{skip}\curvearrowright c|\!\}\ \{\!|\mathsf{input}\ w|\!\} & \text{(CHAM-Read)}\\
\{\!|\texttt{print(}\,i\,\texttt{)}\curvearrowright c|\!\}\ \{\!|\mathsf{output}\ w|\!\}\rightarrow\{\!|\texttt{skip}\curvearrowright c|\!\}\ \{\!|\mathsf{output}\ w:i|\!\} & \text{(CHAM-Print)}\\
\{\!|\texttt{var}\,xl\,;\,s|\!\}\ \{\!|w|\!\}\rightarrow\{\!|s|\!\}\ \{\!|xl\mapsto 0|\!\}\ \{\!|\mathsf{input}\ w|\!\}\ \{\!|\mathsf{output}\ \epsilon|\!\} & \text{(CHAM-Pgm)}
\end{array}
$$

Abrupt Termination

The CHAM semantics of abrupt termination is even more elegant and modular than that using evaluation contexts above, because the other components of the configuration need not be mentioned:

$$i\,/\,0\curvearrowright c\ \ \rightarrow\ \ \texttt{skip} \qquad \text{(CHAM-Div-By-Zero)}$$

$$\texttt{halt}\curvearrowright c\ \ \rightarrow\ \ \texttt{skip} \qquad \text{(CHAM-Halt)}$$

Dynamic Threads

As stated in Section 3.8, the CHAM has been specifically proposed as a model of concurrent computation, based on the chemical metaphor that molecules in solutions can get together and react, with possibly many reactions taking place concurrently. Since there was no concurrency so far in our language, the actual strength of the CHAM has not been seen yet. Recall that the configuration of the existing CHAM semantics of IMP consists of one top-level solution, which contains two subsolutions: a syntactic subsolution holding the remainder of the program organized as a molecule sequentializing computation tasks using the special construct $\curvearrowright$; and a state subsolution containing binding molecules, each binding a different program variable to a value. As seen in Figures 3.44 and 3.45, most of the CHAM rules involve only the syntactic molecule. The state subsolution is only mentioned when the language construct involves program variables.

The above suggests that all a `spawn` statement needs to do is to create an additional syntactic subsolution holding the spawned statement, letting the newly created subsolution molecule to float together with the original syntactic molecule in the same top-level solution. Minimalistically, this can be achieved with the following CHAM rule (which does not consider thread termination yet):

$$\{\!|\, \texttt{spawn}\ s \curvearrowright c |\!\} \to \{\!|\, \texttt{skip} \curvearrowright c |\!\}\ \{\!| s |\!\}$$

Since the order of molecules in a solution is irrelevant, the newly created syntactic molecule has the same rights as the original molecule in reactions involving the state molecule. We can rightfully think of each syntactic subsolution as an independently running thread. The same CHAM rules we had before (see Figures 3.44 and 3.45) can now also apply to the newly created threads. Moreover, reactions taking place only in the syntactic molecules, which are a majority by a large number, can apply truly concurrently. For example, a thread may execute a loop unrolling step while another thread may concurrently perform an addition. The only restriction regarding concurrency is that rule instances must involve disjoint molecules in order to proceed concurrently. That means that it is also possible for a thread to read or write the state while another thread, truly concurrently, performs a local computation. This degree of concurrency was not possible within the other semantic approaches discussed so far in this chapter.

The rule above only creates threads. It does not collect threads when they complete their computation. One could do that with the simple solution-dissolving rule

$$\{\!|\, \texttt{skip} \,|\!\} \to \cdot$$

but the problem is that such a rule cannot distinguish between the original thread and the others, so it would also dissolve the original thread when it completes. However, recall that our design decision for each IMP language extension was to always terminate the program normally, no matter whether it uses the new features or not. The IMP normal result configurations contain a syntactic solution and a state solution, the former holding only `skip`. To achieve that, we can flag the newly created threads for collection as below. Here is our complete CHAM semantics of `spawn`:

$$\mathit{Molecule}\ ::=\ \ldots\ |\ \texttt{die}$$

$$\{\!|\, \texttt{spawn}\ s \curvearrowright c |\!\} \to \{\!|\, \texttt{skip} \curvearrowright c |\!\}\ \{\!| s \curvearrowright \texttt{die} \,|\!\} \qquad \text{(CHAM-Spawn)}$$

$$\{\!|\, \texttt{skip} \curvearrowright \texttt{die} \,|\!\} \to \cdot \qquad \text{(CHAM-Die)}$$

We conclude this section with a discussion on the concurrency of the CHAM above. As already argued, it allows for truly concurrent computations to take place, provided that their corresponding CHAM rule instances do not overlap. While this already goes far beyond the other semantical approaches in terms of concurrency, it still enforces interleaving where it should not. Consider, for example, a global configuration in which two threads are about to lookup two different variables in the state cell. Even though there are good reasons to allow the two threads to proceed concurrently, the CHAM above will not, because the two rule instances (of the same CHAM lookup rule) overlap on the state molecule. This problem can be ameliorated, to some extent, by changing the structure of the top-level configuration to allow all the variable binding molecules currently in the state subsolution to instead float in the top-level solution at the same level with the threads: this way, each thread can independently grab the binding it is interested in without blocking the state anymore. Unfortunately, this still does not completely solve the true concurrency problem, because one could argue that different threads should also be allowed to concurrently read the same variable. Thus, no matter where the binding of that variable is located, the two rule instances cannot proceed concurrently. Moreover, flattening all the syntactic and the semantic ingredients in a top level solution, as the above "fix" suggests, does not scale. Real-life languages can have many configuration items of various kinds, such as, environments, heaps, function/exception/loop stacks, locks held, and so on. Collapsing the contents of all these items in one flat solution would not only go against the CHAM philosophy, but it would also make it hard to understand and control. The K framework (see Section 5) solves this problem by allowing its rules to state which parts of the matched subterm are shared with and which can be concurrently modified by other rules.

Local Variables

The simplest approach to adding blocks with local variables to IMP is to follow and idea similar to the one for reduction semantics with evaluation contexts discussed in Section 3.7.2, assuming the procedure presented in Section 3.5.5 for desugaring blocks with local variables into `let` constructs:

$$\begin{array}{ll}\texttt{let}\,x\,\texttt{=}\,a\,\texttt{in}\,s \curvearrowright c \rightleftharpoons a \curvearrowright \texttt{let}\,x\,\texttt{=}\,\square\,\texttt{in}\,s \curvearrowright c & \\ \{\!|\texttt{let}\,x\,\texttt{=}\,i\,\texttt{in}\,s \curvearrowright c|\!\}\ \{\!|\sigma|\!\} \rightarrow \{\!|s\,\texttt{;}\,x\,\texttt{:=}\,\sigma(x) \curvearrowright c|\!\}\ \{\!|\sigma[i/x]|\!\} & \text{(CHAM-L}\text{ET)}\end{array}$$

As it was the case in Section 3.7.2, the above is going to be problematic when we add `spawn` to the language, too. However, recall that in this language experiment we pretend each language extension is final, in order to understand the modularity and flexibility to change of each semantic approach.

The approach above is very syntactic in nature, following the intuitions of evaluation contexts. In some sense, the above worked because we happened to have an assignment statement in our language, which we used for recovering the value of the bound variable. Note, however, that several computational steps were wasted because of the syntactic translations. What we would have really liked to say is "`let`*Id*`=`*Int*`in`*Context* is a special evaluation context where the current state is updated with the binding whenever is passed through top-down, and where the state is recovered whenever is passed through bottom-up". This was not possible to say with evaluation contexts. When using the CHAM, we are free to disobey the syntax. For example, the alternative definition below captures the essence of the problem and wastes no steps:

$$\begin{array}{l}\texttt{let}\,x\,\texttt{=}\,a\,\texttt{in}\,s \curvearrowright c \rightleftharpoons a \curvearrowright \texttt{let}\,x\,\texttt{=}\,\square\,\texttt{in}\,s \curvearrowright c \\ \{\!|\texttt{let}\,x\,\texttt{=}\,i\,\texttt{in}\,s \curvearrowright c|\!\}\ \{\!|\sigma|\!\} \rightharpoonup \{\!|s \curvearrowright \texttt{let}\,x\,\texttt{=}\,\sigma(x)\,\texttt{in}\,\square \curvearrowright c|\!\}\ \{\!|\sigma[i/x]|\!\} \\ \{\!|\texttt{skip} \curvearrowright \texttt{let}\,x\,\texttt{=}\,v\,\texttt{in}\,\square \curvearrowright c|\!\}\ \{\!|\sigma|\!\} \rightharpoondown \{\!|\,\texttt{skip} \curvearrowright c|\!\}\ \{\!|\sigma[v/x]|\!\}\end{array}$$

In words, the `let` is first heated/cooled in the binding expression. Once that becomes an integer, the `let` is then only heated in its body statement, at the same time updating the state molecule with the binding and storing the return value in the residual `let` construct. Once the `let` body statement becomes `skip`, the solution is cooled down by discarding the residual `let` construct and recovering the state appropriately (we used a "value" v instead of an integer i in the latter rule to indicate the fact that v can also be $\bot$).

Of course, the substitution-based approach discussed in detail in Sections 3.5.6 and 3.7.2 can also be adopted here if one is willing to pay the price for using it:

$$\begin{array}{ll} \texttt{let}\,x\,\texttt{=}\,a\,\texttt{in}\,s \curvearrowright c \rightleftharpoons a \curvearrowright \texttt{let}\,x\,\texttt{=}\,\square\,\texttt{in}\,s \curvearrowright c & \\ \{\!|\texttt{let}\,x\,\texttt{=}\,i\,\texttt{in}\,s \curvearrowright c|\!\}\ \{\!|\sigma|\!\} \to \{\!|s[x'/x] \curvearrowright c|\!\}\ \{\!|\sigma[i/x']|\!\} & \text{where } x' \text{ is a fresh variable} \end{array}$$

Putting Them All Together

Putting together all the language features defined in CHAM above is a bit simpler and more modular than in MSOS (see Section 3.6.2): all we have to do is to take the union of all the syntax and semantics of all the features, removing the original rule for the initialization of the solution (that rule was already removed as part of the addition of input/output to IMP); in MSOS, we also had to add the halting attribute to the labels, which we do not have to do in the case of the CHAM.

Unfortunately, like in the case of the reduction semantics with evaluation contexts of IMP++ in Section 3.7.2, the resulting language is flawed. Indeed, a thread spawned from inside a `let` would be created its own molecule in the top-level solution, which would execute concurrently with all the other execution threads, including its parent. Thus, there is the possibility that the parent will advance to the assignment recovering the value of the `let`-bound variable before the spawned thread terminates, in which case the bound variable would be changed in the state by the parent thread, "unexpectedly" for the spawned thread. One way to address this problem is to rename the bound variable into a fresh variable within the `let` body statement, like we did above, using a substitution operation. Another is to split the state into an environment mapping variables to locations and a store mapping locations to values, and to have each thread consist of a solution holding both its code and its environment. Both these solutions were also suggested in Section 3.5.6, when we discussed how to make small-step SOS correctly capture all the behaviors of the resulting IMP++.

3.8.3 CHAM in Rewriting Logic

As explained above, CHAM rewriting cannot be immediately captured as ordinary rewriting modulo solution multiset axioms such as associativity, commutativity and identity. The distinction between the two arises essentially when a CHAM rule involves only one top-level molecule which is not a solution, because the CHAM laws restrict the applications of such a rule only in solutions while ordinary rewriting allows such rules to apply everywhere. To solve this problem, we wrap each rule in a solution context, that is, we translate CHAM rules of the form

$$m_1\ m_2\ \ldots\ m_k \to m'_1\ m'_2\ \ldots\ m'_l$$

into corresponding rewriting logic rules of the form

$$\{\!|\overline{m_1}\ \overline{m_2}\ \ldots\ \overline{m_k}\ Ms|\!\} \to \{\!|\overline{m'_1}\ \overline{m'_2}\ \ldots\ \overline{m'_l}\ Ms|\!\}$$

where the only difference between the original CHAM terms m_1, m_2, …, m_k, m'_1, m'_2, …, m'_l and their algebraic variants $\overline{m_1}$, $\overline{m_2}$, …, $\overline{m_k}$, $\overline{m'_1}$, $\overline{m'_2}$, …, $\overline{m'_l}$ is that the meta-variables appearing in the former (recall that CHAM rules are rule schematas) are turned into variables of corresponding sorts in the latter, and where Ms is a variable of sort **Bag{***Molecule***}** that does not appear anywhere else in $\overline{m_1}$, $\overline{m_2}$, …, $\overline{m_k}$, $\overline{m'_1}$, $\overline{m'_2}$, …, $\overline{m'_l}$.

With this representation of CHAM rules into rewriting logic rules, it is obvious that rewriting logic's rewriting captures both the Reaction Law and the Chemical Law of the CHAM. What is less obvious is that it also captures the Membrane Law. Indeed, note that the Membrane Law allows rewrites to take place only when the global term is a solution, while rewriting logic allows rewrites to take place anywhere. However, the rewriting logic rule representation above generates only rules that rewrite solutions into solutions. Thus, if the original term to rewrite is a solution, then so it will stay during the entire rewriting process, and so the Membrane Law is also naturally captured by rewriting logic derivations. However, if the original term to rewrite is a proper molecule (which is not a solution), then so it will stay during the entire rewriting logic's rewriting while the CHAM will not reduce it at all. Still it is important to understand that in this case the corresponding rewriting logic theory can perform rewrite steps (in subsolutions of the original term) which are not possible under the CHAM, in particular that it may lead to non-termination in situations where the CHAM is essentially stuck. To reconcile this inherent difference between the CHAM and rewriting logic, we make the reasonable assumption that the original terms to rewrite can only be solutions. Note that the CHAM sequents in Definition 19 already assume that one only derives solution terms.

The only CHAM law which has not been addressed above is the Airlock Law. Rewriting logic has no builtin construct resembling CHAM's airlock, but its multiset matching is powerful enough to allow us to capture the airlock's behavior through rewrite rules. One possibility is to regard the airlock operation like any other molecular construct. Indeed, from a rewriting logic perspective, the Airlock Law says that any molecule inside a solution can be matched and put into an airlock next to the remaining solution wrapped into a membrane, and this process is reversible. This behavior can be achieved through the following two (opposite) rewrite logic rules, where M is a molecule variable and Ms is a bag-of-molecules variable:

$$\begin{array}{rcl} \{\!|M\ Ms|\!\} & \to & \{\!|M \triangleright \{\!|Ms|\!\}|\!\} \\ \{\!|M \triangleright \{\!|Ms|\!\}|\!\} & \to & \{\!|M\ Ms|\!\} \end{array}$$

Another possibility to capture airlock's behavior in rewriting logic is to attempt to eliminate it completely and replace it with matching modulo multiset axioms. While this appears to be possible in many concrete situations, we are however not aware of any general solution to do so systematically for any CHAM. The question is whether the elimination of the airlock is indeed safe, in the sense that the resulting rewriting logic theory does not lose any of the original CHAM's behaviors. One may think that thanks to the restricted form of CHAM's rules, the answer is immediately positive. Indeed, since the CHAM disallows any other constructs for solutions except its builtin membrane operation (a CHAM can only add new syntactic constructs for molecules, but not for solutions) and since solution subterms can either contain only one molecule or otherwise be meta-variables (to avoid multiset matching), we can conclude that in any CHAM rule, a subterm containing an airlock operation at its top can only be of the form $m \triangleright \{\!|m'|\!\}$ or of the form $m \triangleright s$ with s a meta-variable. Both these cases can be uniformly captured as subterms of the form $\overline{m} \triangleright \{\!|ms|\!\}$ with ms a term of sort **Bag{**_Molecule_**}** in rewriting logic, the former by taking $k = 1$ and $ms = m'$ and the latter by replacing the metavariable s with a term of the form $\{\!|Ms|\!\}$ everywhere in the rule, where Ms is a fresh variable of sort **Bag{**_Molecule_**}**. Unfortunately, it is not clear how we can eliminate the airlock from subterms of the form $\overline{m} \triangleright \{\!|ms|\!\}$. If such terms appear in a solution or at the top of the rule, then one can replace them by their corresponding bag-of-molecule terms $\overline{m}\ ms$. However, if they appear in a proper molecule context (i.e., not in a solution context), then they cannot be replaced by $\overline{m}\ ms$ (first, we would get a parsing error by placing a bag-of-molecule term in a molecule place; second, reactions cannot take place in ms anymore, because it is not surrounded by a membrane). We cannot replace $\overline{m} \triangleright \{\!|ms|\!\}$ by $\{\!|\overline{m}\ ms|\!\}$ either, because that would cause a double membrane when the thus modified airlock reaches a solution context. Therefore, we keep the airlock.

sorts:

$\quad$ *Molecule*, *Solution*, $\mathbf{Bag\{}$*Molecule*$\mathbf{\}}$

subsorts:

$\quad$ *Solution* < *Molecule*

$\quad$ // One may also need to subsort to *Molecule* specific syntactic categories (*Int*, *Bool*, etc.)

operations:

$\quad \{\!|\_|\!\} : \mathbf{Bag\{}\mathit{Molecule}\mathbf{\}} \to \mathit{Solution}$ // membrane operator

$\quad \_ \triangleright \_ : \mathit{Molecule} \times \mathit{Solution} \to \mathit{Molecule}$ // airlock operator

$\quad$ // One may also need to define specific syntactic constructs for *Molecule* ($\_ + \_$, $\_ \mapsto \_$, etc.)

rules:

$\quad$ // Add the following two generic (i.e., same for all CHAMs) airlock rewrite logic rules:

$\quad \{\!|M\ Ms|\!\} \leftrightarrow \{\!|M \triangleright \{\!|Ms|\!\}|\!\}$ // M, Ms variables of sorts *Molecule*, $\mathbf{Bag\{}$*Molecule*$\mathbf{\}}$, resp.

$\quad$ // For each specific CHAM rule $m_1\ m_2\ \ldots m_k \to m'_1\ m'_2\ \ldots m'_l$ add a rewriting logic rule

$\quad \{\!|\overline{m_1}\ \overline{m_2}\ \ldots\ \overline{m_k}\ Ms|\!\} \to \{\!|\overline{m'_1}\ \overline{m'_2}\ \ldots\ \overline{m'_l}\ Ms|\!\}$ // Ms variable of sort $\mathbf{Bag\{}$*Molecule*$\mathbf{\}}$

$\quad$ // where $\overline{m}$ replaces each meta-variable in m by a variable of corresponding sort.

Figure 3.47: Embedding of a chemical abstract machine into rewriting logic ($\text{CHAM} \rightsquigarrow \mathcal{R}_{\text{CHAM}}$).

Putting all the above together, we can associate a rewriting logic theory to any CHAM as shown in Figure 3.47; for simplicity, we assumed that the CHAM has only one *Molecule* syntactic category and, implicitly, only one corresponding *Solution* syntactic category. In Figure 3.47 and elsewhere in this section, we use the notation *left* $\leftrightarrow$ *right* as a shorthand for two opposite rewrite rules, namely for both *left* $\to$ *right* and *right* $\to$ *left*. The discussion above implies the following result:

Theorem 10. ***(Embedding of the chemical abstract machine into rewriting logic)*** *If* CHAM *is a chemical abstract machine, sol and sol′ are two solutions, and* $\mathcal{R}_{\text{CHAM}}$ *is the rewrite logic theory associated to* CHAM *as in Figure 3.47, then the following hold:*

1. $\text{CHAM} \vdash sol \to sol'$ *if and only if* $\mathcal{R}_{\text{CHAM}} \vdash \overline{sol} \to^1 \overline{sol'}$;
2. $\text{CHAM} \vdash sol \to^* sol'$ *if and only if* $\mathcal{R}_{\text{CHAM}} \vdash \overline{sol} \to \overline{sol'}$.

Therefore, one-step solution rewriting in $\mathcal{R}_{\text{CHAM}}$ corresponds precisely to one-step solution rewriting in the original CHAM and thus, one can use $\mathcal{R}_{\text{CHAM}}$ as a replacement for CHAM for any reduction purpose. Unfortunately, this translation of CHAM into rewriting logic does not allow us to borrow the latter's concurrency to obtain the desired concurrent rewriting computational mechanism of the former. Indeed, the desired CHAM concurrency says that "different rule instances can apply concurrently in the same solution as far as they act on different molecules". Unfortunately, the corresponding rewrite logic rule instances cannot apply concurrently according to rewriting logic's semantics because both instances would match the entire solution, including the membrane (and rule instances which overlap cannot proceed concurrently in rewriting logic—see Section 2.7).

The CHAM of IMP in Rewriting Logic

Figure 3.48 shows the rewrite theory $\mathcal{R}_{\text{CHAM(IMP)}}$ obtained by applying the generic transformation procedure in Figure 3.47 to the CHAM of IMP discussed in this section and summarized in

sorts:

$\mathit{Molecule}$, $\mathit{Solution}$, $\mathbf{Bag\{}\mathit{Molecule}\mathbf{\}}$ // generic CHAM sorts

subsorts:

$\mathit{Solution} < \mathit{Molecule}$ // generic CHAM subsort

Int, Bool, $\mathit{Id} < \mathit{Molecule}$ // additional IMP-specific syntactic categories

operations:

$\{\!|\_|\!\} : \mathbf{Bag\{}\mathit{Molecule}\mathbf{\}} \to \mathit{Solution}$ // generic membrane

$\_ \triangleright \_ : \mathit{Molecule} \times \mathit{Solution} \to \mathit{Molecule}$ // generic airlock

$\_ \curvearrowright \_ : \mathit{Molecule} \times \mathit{Molecule} \to \mathit{Molecule}$ // "followed by" operator, for evaluation strategies

// Plus all the IMP language constructs and evaluation contexts (all listed in Figure 3.30),

// collapsing all syntactic categories different from *Int*, *Bool* and *Id* into *Molecule*

rules:

// Airlock:

$\{\!|M\ Ms|\!\} \leftrightarrow \{\!|M \triangleright \{\!|Ms|\!\}|\!\}$

// Heating/cooling rules corresponding to the evaluation strategies of IMP's constructs:

$\{\!|(A_1 \texttt{+} A_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(A_1 \curvearrowright \square \texttt{+} A_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(A_1 \texttt{+} A_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(A_2 \curvearrowright A_1 \texttt{+} \square \curvearrowright C)\ Ms|\!\}$

$\{\!|(A_1 \texttt{/} A_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(A_1 \curvearrowright \square \texttt{/} A_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(A_1 \texttt{/} A_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(A_2 \curvearrowright A_1 \texttt{/} \square \curvearrowright C)\ Ms|\!\}$

$\{\!|(A_1 \texttt{<=} A_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(A_1 \curvearrowright \square \texttt{<=} A_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(I_1 \texttt{<=} A_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(A_2 \curvearrowright I_1 \texttt{<=} \square \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{not}\, B \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(B \curvearrowright \texttt{not}\, \square \curvearrowright C)\ Ms|\!\}$

$\{\!|(B_1\, \texttt{and}\, B_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(B_1 \curvearrowright \square\, \texttt{and}\, B_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(X \texttt{:=} A \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(A \curvearrowright X \texttt{:=} \square \curvearrowright C)\ Ms|\!\}$

$\{\!|(S_1 \texttt{;} S_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(S_1 \curvearrowright \square \texttt{;} S_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|S\ Ms|\!\} \leftrightarrow \{\!|(S \curvearrowright \square)\ Ms|\!\}$

$\{\!|(\texttt{if}\, B\, \texttt{then}\, S_1\, \texttt{else}\, S_2 \curvearrowright C)\ Ms|\!\} \leftrightarrow \{\!|(B \curvearrowright \texttt{if}\ \square\ \texttt{then}\, S_1\, \texttt{else}\, S_2 \curvearrowright C)\ Ms|\!\}$

// Semantic rewrite rules corresponding to reaction computational steps

$\{\!|\{\!|X \curvearrowright C\,|\!\}\ \{\!|X \mapsto I \triangleright \sigma|\!\}\ Ms|\!\} \to \{\!|\{\!|I \curvearrowright C\,|\!\}\ \{\!|X \mapsto I \triangleright \sigma|\!\}\ Ms|\!\}$

$\{\!|(I_1 \texttt{+} I_2 \curvearrowright C)\ Ms|\!\} \to \{\!|(I_1 +_{\mathit{Int}} I_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(I_1 \texttt{/} I_2 \curvearrowright C)\ Ms|\!\} \to \{\!|(I_1 /_{\mathit{Int}} I_2 \curvearrowright C)\ Ms|\!\}\ \mathbf{if}\, I_2 \neq 0$

$\{\!|(I_1 \texttt{<=} I_2 \curvearrowright C)\ Ms|\!\} \to \{\!|(I_1 \leq_{\mathit{Int}} I_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{not true} \curvearrowright C)\ Ms|\!\} \to \{\!|(\texttt{false} \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{not false} \curvearrowright C)\ Ms|\!\} \to \{\!|(\texttt{true} \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{true and}\, B_2 \curvearrowright C)\ Ms|\!\} \to \{\!|(B_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{false and}\, B_2 \curvearrowright C)\ Ms|\!\} \to \{\!|(\texttt{false} \curvearrowright C)\ Ms|\!\}$

$\{\!|\{\!|X \texttt{:=} I \curvearrowright C|\!\}\ \{\!|X \mapsto J \triangleright \sigma|\!\}\ Ms|\!\} \to \{\!|\{\!|\texttt{skip} \curvearrowright C|\!\}\ \{\!|X \mapsto I \triangleright \sigma|\!\}\ Ms|\!\}$

$\{\!|(\texttt{skip ;}\, S_2 \curvearrowright C)\ Ms|\!\} \to \{\!|(S_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{if true then}\, S_1\, \texttt{else}\, S_2 \curvearrowright C)\ Ms|\!\} \to \{\!|(S_1 \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{if false then}\, S_1\, \texttt{else}\, S_2 \curvearrowright C)\ Ms|\!\} \to \{\!|(S_2 \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{while}\, B\, \texttt{do}\, S \curvearrowright C)\ Ms|\!\} \to \{\!|(\texttt{if}\, B\, \texttt{then}\, (S\ \texttt{;}\ \texttt{while}\, B\, \texttt{do}\, S)\, \texttt{else skip} \curvearrowright C)\ Ms|\!\}$

$\{\!|(\texttt{var}\, xl\, \texttt{;}\, s)\ Ms|\!\} \to \{\!|\{\!|s|\!\}\ \{\!|xl \mapsto 0|\!\}\ Ms|\!\}$

// State initialization:

$\{\!|(X, Xl \mapsto I)\ Ms|\!\} \to \{\!|(X \mapsto I \triangleright \{\!|Xl \mapsto I|\!\})\ Ms|\!\}$

Figure 3.48: $\mathcal{R}_{\mathrm{CHAM(IMP)}}$: The CHAM of IMP in rewriting logic.

Figures 3.44 and 3.45. In addition to the generic CHAM syntax, as indicated in the comment under subsorts in Figure 3.47 we also subsort the builtin sorts of IMP, namely *Int*, *Bool* and *Id*, to *Molecule*. The "followed by" $\_ \curvearrowright \_$ construct for molecules is necessary for defining the evaluation strategies of the various IMP language constructs as explained above; we believe that some similar operator is necessary when defining any programming language whose constructs have evaluation strategies because, as explained above, it appears that the CHAM airlock operator is not suitable for this task. For notational simplicity, we stick to our previous convention that $\_ \curvearrowright \_$ is right associative and binds less tight than any other molecular construct. Finally, we add molecular constructs corresponding to all the syntax that we need in order to define the IMP semantics, which includes syntax for language constructs, for evaluation contexts, and for the state and state initialization.

The rewrite rules in Figure 3.48 are straightforward, following the transformation described in Figure 3.47. We grouped them in four categories: (1) the airlock rule is reversible and precisely captures the Airlock Law of the CHAM; (2) the heating/cooling rules, also reversible, capture the evaluation strategies of IMP's language constructs (see Figure 3.44); (3) the semantic rules are irreversible and capture the computational steps of the IMP semantics (see Figure 3.45); (4) the state initialization rule corresponds to the heating rule in Figure 3.45.

Our CHAM-based rewriting logic semantics of IMP in Figure 3.48 follows blindly the CHAM of IMP in Figures 3.44 and 3.45. All it does is to mechanically apply the transformation in Figure 3.47, without making any attempts to optimize the resulting rewriting logic theory. For example, it is easy to see that the syntactic molecules will always contain only one molecule. Also, it is easy to see that the state molecule can be initialized in such a way that at any moment during the initialization rewriting sequence any subsolution containing a molecule of the form $xl \mapsto i$, with xl a proper list, will contain no other molecule. Finally, one can also notice that the top level solution will always contain only two molecules, namely what we called a syntactic solution and a state solution. All these observations suggest that we can optimize the rewriting logic theory in Figure 3.48 by deleting the variable *Ms* from every rule except the airlock ones. While one can do that for our simple IMP language here, one has to be careful with such optimizations in general. For example, we later on add threads to IMP (see Section 3.5), which implies that the top level solution will contain a dynamic number of syntactic subsolutions, one per thread; if we remove the *Ms* variable from the lookup and assignment rules in Figure 3.48, then those rules will not work whenever there are more than two threads running concurrently. On the other hand, the *Ms* variable in the rule for variable declarations can still be eliminated. The point is that one needs to exercise care when one attempts to hand-optimize the rewrite logic theories resulting from mechanical semantic translations.

☆ The CHAM of IMP in Maude

Like for the previous semantics, it is relatively straightforward to mechanically translate the corresponding rewrite theories into Maude modules. However, unlike in the previous semantics, the resulting Maude modules are not immediately executable. The main problem is that, in spite of its elegant chemical metaphor, the CHAM was not conceived to be blindly executable. For example, most of the heating and cooling rules tend to be reversible, leading to non-termination of the underlying rewrite relation. Non-termination is not a problem per se in rewriting logic and in Maude, because one can still use other formal analysis capabilities of these such as search and model checking, but from a purely pragmatic perspective it is rather inconvenient not to be able to execute an operational semantics of a language, particularly of a simple one like our IMP. Moreover, since the state space of a CHAM can very quickly grow to unmanageable sizes even when the state

```
mod CHAM is
  sorts Molecule Solution Bag{Molecule} .
  subsort Solution < Molecule < Bag{Molecule} .
  op empty : -> Bag{Molecule} .
  op __ : Bag{Molecule} Bag{Molecule} -> Bag{Molecule} [assoc comm id: empty] .
  op {|_|} : Bag{Molecule} -> Solution .
  op _<|_ : Molecule Solution -> Molecule [prec 110] .
  var M : Molecule .  var Ms : Bag{Molecule} .
  rl {| M Ms |} => {| M <| {| Ms |} |} .
  rl {| M <| {| Ms |} |} => {| M Ms |} .
endm

mod IMP-CHAM-SYNTAX is including PL-INT + CHAM .
  subsort Int < Molecule .
--- Define all the IMP constructs as molecule constructs
  op _+_ : Molecule Molecule -> Molecule [prec 33 gather (E e) format (d b o d)] .
  op _/_ : Molecule Molecule -> Molecule [prec 31 gather (E e) format (d b o d)] .
--- ... and so on
--- Add the hole as basic molecular construct, to allow for building contexts as molecules
  op [] : -> Molecule .
endm

mod IMP-HEATING-COOLING-CHAM-FAILED-1 is including IMP-CHAM-SYNTAX .
  var A1 A2 : Molecule .  var Ms : Bag{Molecule} .
--- + strict in its first argument
  rl {| (A1 + A2) Ms |} => {| (A1 <| {| [] + A2 |}) Ms |} .
  rl {| (A1 <| {| [] + A2 |}) Ms |} => {| (A1 + A2) Ms |} .
--- / strict in its second argument
  rl {| (A1 / A2) Ms |} => {| (A2 <| {| A1 / [] |}) Ms |} .
  rl {| (A2 <| {| A1 / [] |}) Ms |} => {| (A1 / A2) Ms |} .
--- ... and so on
endm

mod IMP-HEATING-COOLING-CHAM-FAILED-2 is including IMP-CHAM-SYNTAX .
  var A1 A2 : Molecule .  var Ms : Bag{Molecule} .  var C : Solution .
--- + strict in its first argument
  rl {| (A1 + A2 <| C) Ms |} => {| (A1 <| {| {| [] + A2 <| C |} |}) Ms |} .
  rl {| (A1 <| {| {| [] + A2 <| C |} |}) Ms |} => {| (A1 + A2 <| C) Ms |} .
--- / strict in its second argument
  rl {| (A1 / A2 <| C) Ms |} => {| (A2 <| {| {| A1 / [] <| C |} |}) Ms |} .
  rl {| (A2 <| {| {| A1 / [] <| C |} |}) Ms |} => {| (A1 / A2 <| C) Ms |} .
--- ... and so on
endm
```

Figure 3.49: Failed attempts to represent the CHAM of IMP in Maude using the airlock mechanism to define evaluation strategies. This figure also highlights the inconvenience of redefining IMP's syntax (the module IMP-CHAM-SYNTAX needs to redefine the IMP syntax as molecule constructs).

space of the represented program is quite small, a direct representation of a CHAM in Maude can easily end up having only a theoretical relevance.

Before addressing the non-termination issue, it is instructive to discuss how a tool like Maude can help us pinpoint and highlight potential problems in our definitions. For example, we have previously seen how we failed, in two different ways, to use CHAM's airlock operator to define the evaluation strategies of the various IMP language constructs. We have noticed those problems with using the airlock for evaluation strategies by actually experimenting with CHAM definitions in Maude, more precisely by using Maude's search command to explore different behaviors of a program. We next discuss how one can use Maude to find out that both our attempts to use airlock for evaluation strategies fail. Figure 3.49 shows all the needed modules. `CHAM` defines the generic syntax of the chemical abstract machine together with its airlock rules in Maude, assuming only one type of molecule. `IMP-CHAM-SYNTAX` defines the syntax of IMP as well as the syntax of IMP's evaluation contexts as a syntax for molecules; since there is only one syntactic category for syntax now, namely `Molecule`, adding $\square$ as a `Molecule` constant allows for `Molecule` to also include all the IMP evaluation contexts, as well as many other garbage terms (e.g., $\square + \square$, etc.). The module `IMP-HEATING-COOLING-CHAM-FAILED-1` represents in Maude, using the general translation of CHAM rules into rewriting logic rules shown in Figure 3.47, heating/cooling rules of the form

$$a_1 + a_2 \quad \rightleftharpoons \quad a_1 \triangleleft \{\!| \square + a_2 |\!\}$$

Only two such groups of rules are shown, which is enough to show that we have a problem. Indeed, all four Maude search commands below succeed:

```
search[1] {| 3 / (1 + 2) |} =>* {| 1 ([] + 2) (3 / []) |} .
search[1] {| 1 ([] + 2) (3 / []) |} =>* {| 3 / (1 + 2) |} .
search[1] {| (3 / 1) + 2 |} =>* {| 1 ([] + 2) (3 / []) |} .
search[1] {| 1 ([] + 2) (3 / []) |} =>* {| (3 / 1) + 2 |} .
```

That means that the Maude solution terms `{| 3 / (1 + 2) |}` and `{| (3 / 1) + 2 |}` can rewrite into each other, which is clearly wrong. Similarly, `IMP-HEATING-COOLING-CHAM-FAILED-2` represents in Maude heating/cooling rules of the form

$$(a_1 + a_2) \triangleleft c \quad \rightleftharpoons \quad a_1 \triangleleft \{\!|\{\!|(\square + a_2) \triangleleft c|\!\}|\!\}$$

One can now check that this second approach gives us what we wanted, that is, a chemical representation of syntax where each subsolution directly contains no more than one $\square$:

```
search[1] {| 3 / (1 + 2) |} =>* {| 1 {| ([] + 2) {| 3 / [] |} |} |} .
search[1] {| (3 / 1) + 2 |} =>* {| 1 {| (3 / []) {| [] + 2 |} |} |} .
```

Indeed, both Maude search commands above succeed. Unfortunately, the following commands

```
search[1] {| 3 / (1 + 2) |} =>* {| 1 {| [] + 2 |} {| 3 / [] |} |} .
search[1] {| 1 {| [] + 2 |} {| 3 / [] |} |} =>* {| 3 / (1 + 2) |} .
search[1] {| (3 / 1) + 2 |} =>* {| 1 {| [] + 2 |} {| 3 / [] |} |} .
search[1] {| 1 {| [] + 2 |} {| 3 / [] |} |} =>* {| (3 / 1) + 2 |} .
```

also succeed, showing that our second attempt to use the airlock for evaluation strategies fails, too.

One could also try the following, which should also succeed (proof is the searches above):

```
search[1] {| 3 / (1 + 2) |} =>* {| (3 / 1) + 2 |} .
search[1] {| (3 / 1) + 2 |} =>* {| 3 / (1 + 2) |} .
```

```
mod CHAM is
  sorts Molecule Solution Bag{Molecule} .
  subsort Solution < Molecule < Bag{Molecule} .
  op empty : -> Bag{Molecule} .
  op __ : Bag{Molecule} Bag{Molecule} -> Bag{Molecule} [assoc comm id: empty] .
  op {|_|} : Bag{Molecule} -> Solution .
  op _<|_ : Molecule Solution -> Molecule [prec 110] .
--- The airlock is unnecessary in this particular example.
--- We keep it, though, just in case will be needed as we extend the language.
--- We comment the two airlock rules below out to avoid non-termination.
--- Otherwise we would have to use slower search comments instead of rewrite.
---  var M : Molecule .  var Ms : Bag{Molecule} .
---  rl {| M Ms |} => {| M <| {| Ms |} |} .
---  rl {| M <| {| Ms |} |} => {| M Ms |} .
endm
```

Figure 3.50: Generic representation of the CHAM in Maude.

However, on our machine (Linux, 2.4GHz, 8GB memory) Maude 2 ran out of memory after several minutes when asked to execute any of the two search commands above.

Figures 3.50, 3.51 and 3.52 give a correct Maude representation of CHAM(IMP), based on the rewrite logic theory $\mathcal{R}_{\text{CHAM(IMP)}}$ in Figure 3.48. Three important observations were the guiding factors of our Maude semantics of CHAM(IMP):

1. While the approach to syntax in Figure 3.49 elegantly allows to include the syntax of evaluation contexts into the syntax of molecules by simply defining $\square$ as a molecular construct, unfortunately it still requires us to redefine the entire syntax of IMP into specific constructs for molecules. This is inconvenient at best. We can do better by simply subsorting to `Molecule` all those syntactic categories that the approach in Figure 3.49 would collapse into `Molecule`. This way, any fragment of IMP code parses to a subsort of `Molecule`, so in particular to `Molecule`. Unfortunately, evaluation contexts are now ill-formed; for example, $\square + 2$ attempts to sum a molecule with an integer, which does not parse. To fix this, we define a new sort for the $\square$ constant, say `Hole`, and declare it as a subsort of each IMP syntactic category that is subsorted to `Molecule`. In particular, `Hole` is a subsort of `AExp`, so $\square + 2$ parses to `AExp`.

2. As discussed above, most of CHAM's heating/cooling rules are reversible and thus, when regarded as rewrite rules, lead to non-termination. To avoid non-termination, we restrict the heating/cooling rules so that heating only applies when the heated subterm has computational contents (i.e., it is not a result) while cooling only applies when the cooled term is completely processed (i.e., it is a result). This way, the heating and the cooling rules are applied in complementary situations, in particular they are not reversible anymore, thus avoiding non-termination. To achieve this, we introduce a subsort `Result` of `Molecule`, together with subsorts of it corresponding to each syntactic category of IMP as well as with an explicit declaration of IMP's results as appropriate terms of corresponding result sort.

3. The CHAM's philosophy is to represent data, in particular program states, using its builtin support for solutions as multisets of molecules, and to use airlock operations to extract pieces of data from such solutions whenever needed. This philosophy is justified both by chemical and by mathematical intuitions, namely that one needs an additional step to observe inside a

```
mod IMP-HEATING-COOLING-CHAM is including IMP-SYNTAX + CHAM .
  sorts Hole ResultAExp ResultBExp ResultStmt Result .
  subsorts Hole < AExp BExp Stmt < Molecule .
  subsorts ResultAExp ResultBExp ResultStmt < Result < Molecule .
  subsorts Int < ResultAExp < AExp .
  subsorts Bool < ResultBExp < BExp .
  subsorts ResultStmt < Stmt .
  op skip : -> ResultStmt [ditto] .

  op [] : -> Hole .
  op _~>_ : Molecule Molecule -> Molecule [gather(e E) prec 120] .

  var X : Id .  var C : [Molecule] .  var A A1 A2 : AExp .  var R R1 R2 : Result .
  var B B1 B2 : BExp .  var I I1 I2 : Int .  var S S1 S2 : Stmt .  var Ms : Bag{Molecule} .

 crl {| (A1 + A2 ~> C) Ms |} => {| (A1 ~> [] + A2 ~> C) Ms |} if notBool(A1 :: Result) .
  rl {| (R1 ~> [] + A2 ~> C) Ms |} => {| (R1 + A2 ~> C) Ms |} .

 crl {| (A1 + A2 ~> C) Ms |} => {| (A2 ~> A1 + [] ~> C) Ms |} if notBool(A2 :: Result) .
  rl {| (R2 ~> A1 + [] ~> C) Ms |} => {| (A1 + R2 ~> C) Ms |} .

 crl {| (A1 / A2 ~> C) Ms |} => {| (A1 ~> [] / A2 ~> C) Ms |} if notBool(A1 :: Result) .
  rl {| (R1 ~> [] / A2 ~> C) Ms |} => {| (R1 / A2 ~> C) Ms |} .

 crl {| (A1 / A2 ~> C) Ms |} => {| (A2 ~> A1 / [] ~> C) Ms |} if notBool(A2 :: Result) .
  rl {| (R2 ~> A1 / [] ~> C) Ms |} => {| (A1 / R2 ~> C) Ms |} .

 crl {| (A1 <= A2 ~> C) Ms |} => {| (A1 ~> [] <= A2 ~> C) Ms |} if notBool(A1 :: Result) .
  rl {| (R1 ~> [] <= A2 ~> C) Ms |} => {| (R1 <= A2 ~> C) Ms |} .

 crl {| (R1 <= A2 ~> C) Ms |} => {| (A2 ~> R1 <= [] ~> C) Ms |} if notBool(A2 :: Result) .
  rl {| (R2 ~> R1 <= [] ~> C) Ms |} => {| (R1 <= R2 ~> C) Ms |} .

 crl {| (not B ~> C) Ms |} => {| (B ~> not [] ~> C) Ms |} if notBool(B :: Result) .
  rl {| (R ~> not [] ~> C) Ms |} => {| (not R ~> C) Ms |} .

 crl {| (B1 and B2 ~> C) Ms |} => {| (B1 ~> [] and B2 ~> C) Ms |} if notBool(B1 :: Result) .
  rl {| (R1 ~> [] and B2 ~> C) Ms |} => {| (R1 and B2 ~> C) Ms |} .

 crl {| (X := A ~> C) Ms |} => {| (A ~> X := [] ~> C) Ms |} if notBool(A :: Result) .
  rl {| (R ~> X := [] ~> C) Ms |} => {| (X := R ~> C) Ms |} .

 crl {| (S1 ; S2 ~> C) Ms |} => {| (S1 ~> [] ; S2 ~> C) Ms |} if notBool(S1 :: Result) .
  rl {| (R1 ~> [] ; S2 ~> C) Ms |} => {| (R1 ; S2 ~> C) Ms |} .

 crl {| S Ms |} => {| (S ~> []) Ms |} if notBool(S :: Result) .
  rl {| (R ~> []) Ms |} => {| R Ms |} .

 crl {| (if B then S1 else S2 ~> C) Ms |} => {| (B ~> if [] then S1 else S2 ~> C) Ms |}
  if notBool(B :: Result) .
  rl {| (R ~> if [] then S1 else S2 ~> C) Ms |} => {| (if R then S1 else S2 ~> C) Ms |} .
endm
```

Figure 3.51: Efficient heating-cooling rules for IMP in Maude.

```
mod IMP-SEMANTICS-CHAM is including IMP-HEATING-COOLING-CHAM + STATE .
  subsort Pgm State < Molecule .
  var X : Id .  var Xl : List{Id} .  var C : Molecule .  var Ms : Bag{Molecule} .
  var Sigma : State .  var B B2 : BExp .  var I J I1 I2 : Int .  var S S1 S2 : Stmt .
  rl {| {| X ~> C |} {| X |-> I & Sigma |} Ms |} => {| {| I ~> C |} {| X |-> I & Sigma |} Ms |} .
  rl {| (I1 + I2 ~> C) Ms |} => {| (I1 +Int I2 ~> C) Ms |} .
 crl {| (I1 / I2 ~> C) Ms |} => {| (I1 /Int I2 ~> C) Ms |} if I2 =/=Bool 0 .
  rl {| (I1 <= I2 ~> C) Ms |} => {| (I1 <=Int I2 ~> C) Ms |} .
  rl {| (not true ~> C) Ms |} => {| (false ~> C) Ms |} .
  rl {| (not false ~> C) Ms |} => {| (true ~> C) Ms |} .
  rl {| (true and B2 ~> C) Ms |} => {| (B2 ~> C) Ms |} .
  rl {| (false and B2 ~> C) Ms |} => {| (false ~> C) Ms |} .
  rl {| {| X := I ~> C |} {| X |-> J & Sigma |} Ms |} => {| {| skip ~> C |} {| X |-> I & Sigma |} Ms |} .
  rl {| (skip ; S2 ~> C) Ms |} => {| (S2 ~> C) Ms |} .
  rl {| (if true  then S1 else S2 ~> C) Ms |} => {| (S1 ~> C) Ms |} .
  rl {| (if false then S1 else S2 ~> C) Ms |} => {| (S2 ~> C) Ms |} .
  rl {| (while B do S ~> C) Ms |} => {| (if B then S ; while B do S else skip ~> C) Ms |} .
  rl {| (var Xl ; S) Ms |} => {| {| S |} {| Xl |-> 0 |} Ms |} .
endm
```

Figure 3.52: The CHAM of IMP in Maude.

solution and, respectively, that multiset matching is a complex operation (it is actually an intractable problem) whose complexity cannot be simply "swept under the carpet". While we agree with these justifications for the airlock operator, one should also note that impressive progress has been made in the last two decades, after the proposal of the chemical abstract machine, in terms of multiset matching. For example, languages like Maude build upon very well-engineered multiset matching techniques. We believe that these recent developments justify us, at least in practical language definitions, to replace the expensive airlock operation of the CHAM with the more available and efficient multiset matching of Maude.

Figure 3.50 shows the generic CHAM syntax that we extend in order to define CHAM(IMP). Since we use Maude's multiset matching instead of state airlock and since we cannot use the airlock for evaluation strategies either, there is effectively no need for the airlock in our Maude definition of CHAM(IMP). Moreover, since the airlock rules are reversible, their introduction would yield non-termination. Consequently, we have plenty of reasons to eliminate them, which is reflected in our `CHAM` module in Figure 3.50. The Maude module in Figure 3.51 defines the evaluation strategies of IMP's constructs and should be now clear: in addition to the syntactic details discussed above, it simply gives the Maude representation of the heating/cooling rules in Figure 3.48. Finally, the Maude module in Figure 3.52 implements the semantic rules and the state initialization heating rule in Figure 3.48, replacing the state airlock operation by multiset matching.

To test the heating/cooling rules, one can write Maude commands such as the two search commands below, asking Maude to search for all heatings of a given syntactic molecule (the sort of X, *Id*, is already declared in the last module):

```
search {| X := 3 / (X + 2) |} =>! Sol:Solution .
search {| X := (Y:Id + Z:Id) / (X + 2) |} =>! Sol:Solution .
```

The former gives only one solution, because heating can only take place on non-result subexpressions

```
Solution 1 (state 4)
states: 5  rewrites: 22 in ... cpu (... real) (... rewrites/second)
Sol:Solution --> {| X ~> [] + 2 ~> 3 / [] ~> X := [] ~> [] |}
```

but the latter gives three solutions:

```
Solution 1 (state 5)
states: 8  rewrites: 31 in ... cpu (... real) (... rewrites/second)
Sol:Solution --> {| Y:Id ~> [] + Z:Id ~> [] / (X + 2) ~> X := [] ~> [] |}

Solution 2 (state 6)
states: 8  rewrites: 31 in ... cpu (... real) (... rewrites/second)
Sol:Solution --> {| Z:Id ~> Y:Id + [] ~> [] / (X + 2) ~> X := [] ~> [] |}

Solution 3 (state 7)
states: 8  rewrites: 31 in ... cpu (... real) (... rewrites/second)
Sol:Solution --> {| X ~> [] + 2 ~> (Y:Id + Z:Id) / [] ~> X := [] ~> [] |}
```

Each of the solutions represents a completely heated term (we used `=>!` in the search commands) and corresponds to a particular order of evaluation of the subexpression in question. The three solutions of the second search command above reflect all possible orders of evaluation allowed by our Maude semantics of CHAM(IMP). Interestingly, note that there is no order in which X is looked up in between Y and Z. This is not a problem for our simple IMP language in this section, but it may result in loss of behaviors when we extend our semantics to IMP++ in Section 3.5. Indeed, it may be that other threads modify the values of X, Y, and Z while the expression above is evaluated by another thread in such a way that behaviors are lost if X is not allowed to be looked up between Y and Z. Consequently, our orientation of the heating/cooling rules came at a price: we lost the fully non-deterministic order of evaluation of the arguments of strict operators; what we obtained is a *non-deterministic choice* evaluation strategy (an order of evaluation is non-deterministically chosen and cannot be changed during the evaluation of the expression—this is discussed in more depth in Section 3.5).

Maude can now act as an execution engine for CHAM(IMP). For example, the Maude command

```
rewrite {| sumPgm |} .
```

where `sumPgm` is the first program defined in the module `IMP-PROGRAMS` in Figure 3.4, produces a result of the form:

```
rewrites: 7343 in ... cpu (... real) (... rewrites/second)
result Solution: {| {| skip |} {| n |-> 0 & s |-> 5050 |} |}
```

Like in the previous Maude semantics, one can also search for all possible behaviors of a program using `search` commands such as

```
search {| sumPgm |} =>! Sol:Solution .
```

Like before, only one behavior will be discovered (IMP is deterministic so far). However, an unexpectedly large number of states is generated, 3918 versus the 1509 states generated by the previous small-step semantics), mainly due to the multiple ways to apply the heating/cooling rules:

```
Solution 1 (state 3918)
states: 3919  rewrites: 12458 in ... cpu (... real) (... rewrites/second)
Sol:Solution --> {| {| skip |} {| n |-> 0 & s |-> 5050 |} |}
```

3.8.4 Notes

The chemical abstract machine, abbreviated CHAM, was introduced by Berry and Boudol in 1990 [10, 11]. In spite of its operational feel, the CHAM should not be mistakenly taken for a variant of (small-step) SOS. In fact, Berry and Boudol presented the CHAM as an *alternative* to SOS, to address a number of limitations inherent to SOS, particularly its lack of true concurrency and what they called SOS' "rigidity to syntax". The basic metaphor giving its name to the CHAM was inspired by Banâtre and Le Mètayer's GAMMA language [5, 6, 7], which was the first to view a distributed state as a solution in which many molecules float, and the first to understand concurrent transitions as reactions that can occur simultaneously in many points of the solution. GAMMA was proposed as a highly-parallel programming language, together with a stepwise program derivation approach that allows to develop provably correct GAMMA programs. However, following the stepwise derivation approach to writing GAMMA programs is not as straightforward as writing CHAM rules. Moreover, CHAM's nesting of solutions allows for structurally more elaborate encodings of data and in particular for more computational locality than GAMMA. Also, the CHAM appears to be more suitable as a framework for defining semantics of programming languages than the GAMMA language; the latter was mainly conceived as a programming language itself, suitable for executing parallel programs on parallel machines [4], rather than as a semantic framework.

The distinction between heating, cooling and reaction rules in CHAM is in general left to the user. There are no well-accepted criteria and/or principles stating when a rule should be in one category or another. For example, should a rule that cleans up a solution by removing residue molecules be a heating or a colling rule? We prefer to think of it as a cooling rule, because it falls under the broad category of rules which "structurally rearrange the solution after reactions take place" which we methodologically decided to call cooling rules. However, the authors of the CHAM prefer to consider such rules to be heating rules, with the intuition that "the residue molecules evaporate when heated" [11]. While the distinction between heating and cooling rules may have a flavor of subjectivity, the distinction between actual reaction rules and heating/cooling rules is more important because it gives the computational granularity of one's CHAM. Indeed, it is common to abstract away the heating/cooling steps in a CHAM rewriting sequence as internal steps and then define various relations of interest on the remaining reaction steps possibly relating different CHAMS, such as behavioral equivalence, simulation and/or bisimulation relations [10, 11].

The technique we used in this section to reversibly and possibly non-deterministically sequentialize the program syntax by heating/cooling it into a list of computational tasks was borrowed from the K framework [78, 76, 51, 74] (see Section 5). This mechanism is also reminiscent of Danvy and Nielsen's refocusing technique [22, 23], used to execute reduction semantics with evaluation contexts by decomposing evaluation contexts into stacks and then only incrementally modifying these stacks during the reduction process. Our representation of the CHAM into rewriting logic was inspired from related representations by Șerbănuță *et al.* [85] and by Meseguer[9] [47]. However, our representation differs in that it enforces the CHAM rewriting to only take place in solutions, this way being completely faithful to the intended meaning of the CHAM reactions, while the representations in [85, 47] are slightly more permissive, allowing rewrites to take place everywhere rules match; as explained, this is relevant only when the left-hand side of the rule contains precisely one molecule.

We conclude this section with a note on the concurrency of CHAM rewriting. As already

[9]Meseguer [47] was published in the same volume of the Journal of Theoretical Computer Science as the CHAM extended paper [11] (the first CHAM paper [10] was published two years before, in 1990, same as the first papers on rewriting logic [45, 44, 46])

explained, we are not aware of any formal definition of a CHAM rewriting relation that captures the truly concurrent computation advocated by the CHAM. This is unfortunate, because its concurrency potential is one of the most appealing aspects of the CHAM. Moreover, as already discussed (after Theorem 10), our rewriting logic representation of the CHAM is faithful only with respect to one non-concurrent step and interleaves steps taking place within the same solutions no matter whether they involve common molecules or not, so we cannot borrow rewriting logic's concurrency to obtain a concurrent semantics for the CHAM. What can be done, however, is to attempt a different representation of the CHAM into rewriting logic based on ideas proposed by Meseguer in [48] to capture (a limited form of) graph rewriting by means of equational encodings. The encoding in [48] is theoretically important, but, unfortunately, yields rewrite logic theories which are not feasible in practice using the current implementation of Maude.

3.8.5 Exercises

Exercise 149. *For any CHAM, any molecules mol_1, ..., mol_k, and any $1 \leq i \leq k$, the sequents*

$$\text{CHAM} \vdash \{\!|mol_1 \ \ldots \ mol_k|\!\} \leftrightarrow \{\!|mol_1 \triangleright \{\!|mol_2 \ \ldots mol_i|\!\} \ mol_{i+1} \ldots mol_k|\!\}$$

are derivable, where if $i = 1$ then the bag $mol_2 \ \ldots mol_i$ is the empty bag.

Exercise 150. *Modify the CHAM semantics of* IMP *in Figure 3.45 to use a state data-structure as we did in the previous semantics instead of representing the state as a solution of binding molecules.*

Exercise 151. *Add a cleanup (cooling) rule to the CHAM semantics of* IMP *in Figure 3.45 to remove the useless syntactic subsolution when the computation is terminated. The resulting solution should only contain a state (i.e., it should have the form $\{\!|\sigma|\!\}$, and not $\{\!|\{\!|\sigma|\!\}|\!\}$ or $\{\!|\{\!|\cdot|\!\} \ \{\!|\sigma|\!\}|\!\}$).*

Exercise 152. *Modify the CHAM semantics of* IMP *in Figures 3.44 and 3.45 so that* / *short-circuits when the numerator evaluates to* 0*.*
<u>Hint:</u> One may need to make one of the heating/cooling rules for / *conditional.*

Exercise 153. *Modify the CHAM semantics of* IMP *in Figures 3.44 and 3.45 so that conjunction is non-deterministically strict in both its arguments.*

Exercise 154. *Same as Exercise 66, but for CHAM instead of big-step SOS: add variable increment to* IMP*, like in Section 3.8.2*
☆ *Like for the Maude definition of big-step SOS and unlike for the Maude definitions of small-step SOS, MSOS and reduction semantics with evaluation contexts above, the resulting Maude definition can only exhibit three behaviors (instead of five) of the program* `nondet++Pgm` *in Exercise 66. This limitation is due to our decision (in Section 3.8) to only heat on non-results and cool on results when implementing CHAMs into Maude. This way, the resulting Maude specifications are executable at the expense at losing some of the behaviors due to non-deterministic evaluation strategies.*

Exercise 155. *Same as Exercise 70, but for the CHAM instead of big-step SOS: add input/output to* IMP*, like in Section 3.8.2.*

Exercise 156. *Same as Exercise 74, but for the CHAM instead of big-step SOS: add abrupt termination to* IMP*, like in Section 3.8.2.*

Exercise 157. *Same as Exercise 85, but for the CHAM instead of small-step SOS: add dynamic threads to* IMP*, like in Section 3.8.2.*

Exercise* 158. *This exercise asks to define* IMP++ *in CHAM, in various ways. Specifically, redo Exercises 95, 96, 97, 98, and 99, but for the CHAM of* IMP++ *discussed in Section 3.8.2 instead of its small-step SOS in Section 3.5.6.*