The Chemical Abstract Machines (CHAM): Solutions to Model Concurrency

Lily Tsai

19 October, 2016

1 The Gamma Language: A Solution Achieving True Concurrency

```
@article{
    Banatre:1990:GMD:113556.113559,
    author = {Banatre, Jean-Pierre and Le Metayer, Daniel},
    title = {The Gamma Model and Its Discipline of Programming},
    journal = {Sci. Comput. Program.},
    issue_date = {Nov. 1990},
    volume = {15},
    number = {1},
    pages = {55--77},
    url = {http://dx.doi.org/10.1016/0167-6423(90)90044-E},
    doi = {10.1016/0167-6423(90)90044-E},
    publisher = {Elsevier North-Holland, Inc.},
    address = {Amsterdam, The Netherlands, The Netherlands},
}
```

Summary. Banatre and Metayer introduce the idea of the Γ language, which is based upon the premise that "concurrent computation should be expressed as 'the global result of the successive applications of local, independent, atomic reactions.'" Γ expresses programs as multisets with transformation rules that alter the components and structure of the multisets. They demonstrate the effectiveness of this new way of reasoning about parallel programs and demonstrate how the Γ model approach entails a novel approach to program design. They also provide a number of small examples and the implementation details of how Γ might be applied.

Evaluation. This paper is a seminal paper because it introduces a completely novel paradigm for thinking about concurrent. Its formalism (the *chemical reaction* model) for expressing programs removes the rigid sequential/geometrical way of viewing concurrency in previous models. Prior to this paper, sequentiality had been inherent in the reasoning about program development; Γ introduces for the first time a complete model that generates systematic way to reason about programs with 'true' concurrency without thinking in an imperative style of programming. This minimal language model has inspired work such as the CHAM and models of software architectures, and has been extended for logic programming, reactive programming, and other applications.

2 The Chemical Abstract Machine (CHAM)

```
@inproceedings{
    bb:chem-machine,
    author = {Gerard Berry and Gerard Boudol},
    title = {The chemical abstract machine},
    booktitle = {Proceedings of the 17th Symposium on POPL},
    year = {1990},
    pages = {81--94},
    location = {San Francisco, California, United States},
    doi = {http://doi.acm.org/10.1145/96709.96717}
}
```

Summary. This paper describes the Chemical Abstract Machine, which takes the Γ model and the chemical metaphor for programs as inspiration for a new way to describe the operational semantics of process calculi. The Chemical Abstract Machine refines and extends the Γ language into a semantic framework. Molecules are given structure as terms and subsolutions are defined using the membrane operator, which allow for more elaborate encodings of data and more computational locality. With these extensions, the CHAM is used to model process calculi such as TCCS, the π-calculus, and encode a higher-order concurrent λ-calculus. The paper shows how the CHAM, compared to the process calculi, allows for a much simpler way to model parallel programs.

Evaluation. The CHAM refines Γ with molecule syntax and additional rules that allow for subsolutions and localized reactions to take place. The locality introduced by the CHAM better represents the idea of several processes running sequential code in parallel (locally, e.g. on a single core), and then communicating with other subsolutions in the complete solution. The CHAM represents a formalization of a way of thinking about concurrency that is radically different from the previously dominant way of thinking about concurrency: process calculi. It shows that process calculi such as CCS only provide synchronous communication (processes sending and receiving must rendezvous), and that a much more natural way of representing the asynchrony desired in parallel programs is through the chemical metaphor (i.e using multisets) and reactions that can occur at any time, given the appropriate molecules. With the CHAM, the chemical metaphor becomes closer to being realized as an actual programming language for parallel programming.

3 Adding Reflexion to the CHAM

```
@inproceedings{
    fg:join-calc,
    author = "Cedric Fournet and Georges Gonthier",
    title = "The Reflexive {CHAM} and the Join-Calculus",
    booktitle = "Proceedings of the 23rd  Symposium on POPL",
    pages = "372--385",
    year = "1996",
    url = "citeseer.ist.psu.edu/fournet95reflexive.html"
}
```

Summary. This paper extends the CHAM with reflexion to provide a better model for distributed settings and asynchronous, distributed, and mobile programming. This enforces locality of reactions (all reactions must occur at a specified reaction site). The paper argues that basing a practical programming languages for distributed environments off the RCHAM is possible. The reflexive CHAM is also described syntactically as a process calculus: the join-calculus. This calculus is shown to be equivalent to the π-calculus.

Evaluation. The RCHAM combines the idea of the CHAM with practical issues such as how the reaction rules and molecules in the solution will be efficiently identified and localized. The paper identifies two large issues with the CHAM, namely the lack of a mixing mechanism ("Brownian motion") that brings molecules together to react, and the complicated pattern matching that might result with many different types of molecules and rules. With these additions, the CHAM becomes a practical model for distributed programming, and shortens the distance between model and program. This paper demonstrates how an idea (the chemical metaphor/CHAM) can be developed so that it can be used for a practical application and deal with implementation realities of distributed programming.

4 JoCaml: The Join Calculus as a practical programming language

```
@techreport{
    mandel:inria-00166125,
    title = {{Programming in JoCaml --- extended version}},
    author = {Mandel, Louis and Maranget, Luc},
    url = {https://hal.inria.fr/inria-00166125},
    type = {research report},
    number = {rr-6261},
    institution = {{inria}},
    year = {2007},
    pdf = {https://hal.inria.fr/inria-00166125/file/rr-6261.pdf},
    hal_id = {inria-00166125},
    hal_version = {v2},
}
```

Summary. This paper describes how the language JoCaml is developed based on the join-calculus. It explains how the OCaml program is extended with join-definitions and how reaction rules and molecules are defined. The paper then demonstrates how an OCaml program can be made into a JoCaml program with distributed and concurrent computations. The paper provides an example application—a ray tracer—and how JoCaml can implement failure detection. Lastly, the paper evaluates the performance of the application implemented in JoCaml.

Evaluation. This paper demonstrates how the idea of the CHAM can be put into practice. It concludes the development cycle of idea (Γ) to model (CHAM) to model refinement (RCHAM) to practical applications (JoCaml).

The History and Future of Chemical Abstract Machines (CHAM)
October 19, 2016
Lily Tsai

---

1 The Γ Language (Banatre & Metayer, 1986)

"Concurrent computation should be expressed as 'the global result of the successive applications of local, independent, atomic reactions'"

Γ is a kernel language (core language with minimal features) which models computation as multiset transforms and draws upon the chemical reaction metaphor. It captures the intuition of computation as a global evolution (i.e. within a 'solution') of a collection of atoms (molecules) which interact (react) freely. Any subset of elements in the multiset can react if they satisfy the reaction condition, and concurrent transitions are non-overlapping reactions can occur in parallel.

- Goal: High-level language that imposes no artificial sequentiality (to be left to the implementation on a particular computation model)
- Imperative and functional approaches need to choose a representation of data (e.g. a set) which then has a hierarchical/recursive structure which the program walks and decomposes to get atomic elements. Γ proposes a program to be understood as a sequence of multiset transforms, which has no hierarchy.

Components of Γ:

- Data Structure: **multiset** ≡ chemical solution
- Execution Rules: **(Reaction Condition, Action)** ≡ (Reactants, Products)
  From examples, they extract five basic programming schemes (mapping an operation over all elements, application of a function to pairs of elements, decomposing elements, selecting elements, optimiser)

Example program using Γ (fib is 0-indexed):

$$
\begin{aligned}
fib(n) &= add(zero(dec(n))) \\
dec &= (\forall x.x > 1, \{x-1, x-2\}) \\
zero &= (\forall x.x = 0, \{1\}) \\
add &= (\forall x, y.True, \{x+y\})
\end{aligned}
$$

Γ has been extended with composition laws, used to program OS kernels and image processing applications, and applied to model process calculi, imperative programming, and software architectures. Today, we explore one particular extension: the Chemical Abstract Machine.

2 The Chemical Abstract Machine (CHAM) Berry & Boudol, 1990

The Chemical Abstract Machine refines and extends the Γ language into a semantic framework. Molecules are given structure as terms and subsolutions are defined using the membrane operator, which allow for more elaborate encodings of data and more computational locality. This allows it to model the semantics of process calculi (CCS and the π-calculus in particular).

(a) Components of CHAM:

$$\begin{aligned} Molecule &::= S \mid Molecule \triangleleft S \\ S &::= \{|\mathbf{Bag}(Molecule)|\} \end{aligned}$$

- $\{|\ .\ |\}$ is the membrane operator that defines a multiset.
- Airlock $m \triangleleft S$ where $\triangleleft$ is the airlock constructor
- $C[]$ is a solution with a hole [] in which to place another molecule

(b) CHAM Rules $S \to S'$

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

where the m_i are not necessarily distinct. Molecules in the rule can only contain subsolution terms which are either solution meta-variables or have the form $\{|m|\}$ where m is a molecule term. This prevents overly complicated pattern matching.

Allowed: $\{|m|\}$ $\{|m \triangleleft S|\}$ $\{|m \triangleleft \{m'\}|\}$

Not Allowed: $\{|m, S|\}$ $\{|m, m'|\}$ $\{|m, m', S|\}$

All rules have no premises.

(c) CHAM General Laws:

- Reaction Law: CHAM rules can only apply in solutions (wrapped by a membrane) and not arbitrarily wherever they match.

$$\textsc{Reaction}\ \frac{m_1, m_2, \cdots, m_k \to m'_1, m'_2, \cdots, m'_l}{\text{CHAM} \vdash \{|M_1, M_2, \cdots, M_k|\} \to \{|M'_1, M'_2, \cdots, M'_l|\}}$$

$M_i,\ M_j$ are instances of $m_i\ , m_j$ by a common substitution.

- Chemistry Law: if a reaction can take place in a certain solution, it can take place in any larger solution.

$$\textsc{Chem}\ \frac{\text{CHAM} \vdash S \to S'}{\text{CHAM} \vdash S \uplus S'' \to S' \uplus S''}\ \uplus \text{ is multiset union}$$

(Combined with Reaction law, this means a CHAM rule can apply inside any solutions having some molecules that match the left-hand side of the rule)

- Membrane Law: a subsolution can evolve freely in any context

$$\textsc{Membrane}\ \frac{\text{CHAM} \vdash S \to S'}{\{\text{CHAM} \vdash |C[S]|\} \to \{|C[S']|\}}$$

- Airlock Law: how to extract or put molecules into a solution (since rule sub-solutions cannot specify a particular molecule term among the rest of the solution)

$$\textsc{Airlock}\ \frac{}{\text{CHAM} \vdash \{|m|\} \uplus S \leftrightarrow \{|m \triangleleft S|\}}$$

Note: The CHAM Laws differ from ordinary rewriting rules because of

i. The natural associativity, commutativity and identity of the molecule grouping (inside a solution) operation

ii. CHAM rules 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 ordinary rewriting rules apply anywhere they match

(d) Rule Categorization

- Reversible Rules (structural rearrangements):
  Heating rules: $\rightharpoonup$ decompose single molecules into simpler ones (hot)
  Cooling rules: $\rightharpoondown$ recompose a compound molecule from its components (frozen)
- Irreversible Reaction rules: $\to$ change the information in a solution in an irreversible way (inert)

(e) Calculus of Communicating Systems (CCS-) Structural Operational Semantics

One of the major motivations of the CHAM was to simplify the expression of concurrent programs from the SOS of CCS and other process calculus. To see the CHAM as a semantic framework, we provide an abstract machine representation of a subset of CCS (CCS-).

CCS- Syntax:

$$\textbf{Agents}\ p ::= 0 \mid \alpha.p \mid (p|p) \mid p\backslash\alpha$$

$$0\ \textbf{is inaction,}\ .\ \textbf{is prefixing,}\ |\ \textbf{is parallel, and}\ \backslash\ \textbf{is restriction}$$

CCS- Semantics have two types of transitions

i. $p \to p'$ (internal actions)

ii. $p \xrightarrow{\alpha} p'$ (where p offers environment action α and becomes p'). α and $\bar{\alpha}$ are the **input** and **output** actions that are required to synchronize communication. Restriction prevents an agent $\alpha.p$ from performing action α if α is the restricted action's name.

$$\textbf{EMIT}\ \frac{}{\alpha.p \xrightarrow{\alpha} p}$$

$$\textbf{PARALLEL}\ \frac{p \to p'}{p|q \to p'|q, \quad q|p \to q|p'}$$

$$\alpha\textbf{-PARALLEL}\ \frac{p \xrightarrow{\alpha} p'}{p|q \xrightarrow{\alpha} p'|q \quad q|p \xrightarrow{\alpha} q|p'}$$

$$\textbf{SYNC}\ \frac{p \xrightarrow{\alpha} p' \quad q \xrightarrow{\bar{\alpha}} q'}{p|q \ \to \ p'|q'}$$

$$\textbf{RESTRICTION}\ \frac{p \to p'}{p\backslash a \to p'\backslash a}$$

$$\alpha\textbf{-RESTRICTION}\ \frac{p \xrightarrow{\alpha} p'}{p\backslash a \xrightarrow{\alpha} p'\backslash a}\ \alpha \notin \{a, \bar{a}\}$$

Example SOS Derivation:

$$\textbf{SYNC}\ \frac{\textbf{EMIT}\ \dfrac{}{a.0 \xrightarrow{a} 0} \qquad \alpha\textbf{-RESTR}\ \dfrac{\textbf{EMIT}\ \dfrac{}{\bar{a}.p \xrightarrow{\bar{a}} p}}{\bar{a}.p|q \xrightarrow{\bar{a}} p|q}\alpha\textbf{-PAR}}{(\bar{a}.p|q)\backslash b \xrightarrow{\bar{a}} (p|q)\backslash b}}{a.0|(\bar{a}.p|q)\backslash b \to 0|(p|q)\backslash b}$$

(A further equivalence rule is required to show that $0|p \equiv p$)

(f) A CHAM for CCS-

$$molecules \equiv \ Agents\ p \ ::= 0 \mid (p|p) \mid \alpha.p \mid p\backslash\alpha$$

Rules:

- Decomposition:

$$p|q \rightleftharpoons p, q$$

- Communication:

$$a.p, \ \bar{a}.q \to p, q$$

- Cleanup: (evaporation)

$$0 \rightharpoonup$$

- Ion release:
$$(\alpha.q)\backslash a \rightleftharpoons \alpha.(q\backslash a) \quad \alpha \notin \{a, \bar{a}\}$$

- Restriction membrane:
$$p\backslash a \rightleftharpoons \{|p|\}\backslash a$$

- Heavy ion release: This rule must work for compound molecules but be expressed in terms of simple molecules/arbitrary solutions. Our first attempt might be
$$\{|\alpha.p, p_1 ...|\} \rightharpoonup \alpha.\{|p, p_1, ...|\}$$
However, this involves complex molecules, is irreversible, and must find an ion within an arbitrary solution. Instead, we use the airlock operator:
$$(\alpha.p) \triangleleft S \rightleftharpoons \alpha.(p \triangleleft S)$$

Example Derivation in CHAM:

$\{\|a.0\|(\bar{a}.p\|q)\backslash b\|\}$	
$\rightharpoonup^* \{\|a.0, \{\|(\bar{a}.p, q)\|\}\backslash b\|\}$	**decomposition, restriction membrane**
$\rightharpoonup \{\|a.0, \{\|(\bar{a}.p \triangleleft \{\|q\|\})\|\}\backslash b\|\}$	**airlock**
$\rightharpoonup \{\|a.0, \{\|\bar{a}.(p \triangleleft \{\|q\|\})\|\}\backslash b\|\}$	**heavy ion**
$\rightharpoonup \{\|a.0, \bar{a}.(p \triangleleft \{\|q\|\})\backslash b\|\}$	**restriction membrane**
$\rightharpoonup \{\|a.0, \bar{a}.((p \triangleleft \{\|q\|\})\backslash b)\|\}$	**ion release**
$\rightarrow \{\|0, (p \triangleleft \{\|q\|\})\backslash b\|\}$	**communication**
$\rightleftharpoons^* \{\|\{\|(p, q)\|\})\backslash b\|\}$	**cleanup, restriction membrane, airlock**

(g) Comparison of CCS- and CHAM

- Internal vs. External: Internal (rewriting) steps can be abstracted away from external (reaction) steps, allowing one to reason about relations only on reaction steps. CCS contains labeled transitions $\xrightarrow{\alpha}$ and internal rules have premises based on external phenomena.
- Lack of Congruence Rules: Multisets are unordered, giving natural associativity and commutativity. SOS requires an extra addition of equivalence to prove commutativity, for example.
- Asynchrony as a primitive: A reaction may execute concurrently and at any time, given that the appropriate molecules are present. Parallel operations in the same solution still continue to evolve. A callback reaction rule can be defined to invoke operations after synchronization of a send/receive occurs.
For example, Sends/Receives can make progress regardless of whether there is a reciprocating call (just 'throw out' the molecule into the solution).

Sync: $send(v)\ \&\ receive() \rightarrow$ **return** v **to** $receive()$
Callback: $receive(v)\ \&\ print() \rightarrow print(v)$
Spawn: $\big(receive() \mid fib() \mid print()\big)\ \&\ \big(send(1) \mid send(2)\big)$

CCS has a 'rigid geometrical vision of concurrency' (i.e. channels and ports) in which components must be brought together to synchronize/rendezvous. Process calculi based off CCS based on atomic non-local interaction and synchronous channels (rendevzous), which introduce global atomic interaction between distant emitters and receivers. In order to send/receive on a channel, global synchronization is necessary to deal with shared, global channel names and coordinate rendezvous between processes listening/sending on the same channel.

The CHAM has been used to simplify reasoning about program execution and program transformations, as is seen in its simulation of CCS, TCCS, π (Milner's calculus of mobile processes), and other representations of of concurrent programs.

3 The Join-Calculus and the Reflexive CHAM

Fournet & Gonthier, 1996

(a) Contribution:

The reflexive CHAM extends the CHAM with reflexion to provide a better model for distributed settings and asynchronous, distributed, and mobile programming. It provides a model that both describes how an implemented programming language should be designed and also reflects the implementation constraints.

The reflexive CHAM can also be described syntactically as a process calculus, called the join-calculus, that is equivalent to Milner's π calculus. Because this calculus has been described with implementation in mind, it has provided the base for practical programming languages.

(b) Motivation

- Why CHAM instead of π-Calculus?
  Implementation constraints of asynchronous systems mean that transmission has to be decoupled from synchronization. The process calculi have two issues: 1) rendezvous and synchronous channels are a primitive, and 2) names are global, meaning that global synchronization is required.
  The CHAM provides a simple, purely asynchronous setting in which the transmission of messages, calls, and returns do not add any contention.
  As we will see, the RCHAM/join calculus makes synchronization behavior explicit, allowing for easy encodings of both asynchronous and synchronous communication.
- Problems with the CHAM:
  The CHAM cites some type of "magical mixing"—Brownian motion in chemistry—that brings molecules close together to react. This random motion has two problems:
  i. A **catalyst** phenomenon (assuming that the chemical rules have disjoint domains), where molecules must travel to a reaction site (associated with their rule) to be sorted, matched, and reacted. Communication is restricted to these sites, which creates a concurrency bottleneck if these sites are few in number and very complex.
  ii. Pattern matching on molecule shapes can grow arbitrarily complicated, making management of each catalyst site (reaction rule) complicated (**clogging** the reaction rule). (This is equivalent to complications of the global shared namespace of the π-calculus).

  The solution that RCHAM proposes is to allow for new reactions (local reactions) to be created with simpler pattern matching for "local" molecules.

(c) Syntax of RCHAM:

Values are names, represented by name variables x, v. $\tilde{x}$ is a tuple of names. $x\langle\tilde{v}\rangle$ is a message: a channel x sends message v.

def D **in** P acts as a **defining molecule** that defines a new reaction D and a molecule P. This adds **reflexion** to the model: reactions can now be dynamically created.

D **or** $J \triangleright P$ represents a reaction that consumes molecules with a specific join pattern J and produces product P. We can think of this as **let** $J =$ **run** P

This corresponds to a transformation rule $(R \vdash M \to (R \vdash M')$.

Definitions act as **catalysts**, where reactants can assemble and react. If a definition is not present (defined), the reaction cannot occur. Thus, reactions are constrained to their local definitions.

$$
\begin{aligned}
P &::= x\langle \tilde{v} \rangle \mid \textbf{def } D \textbf{ in } P \mid (P|P) \\
J &::= x\langle \tilde{v} \rangle \mid (J|J) \\
D &::= J \triangleright P \mid D \wedge D \mid \emptyset \\
\mathit{Solutions} &::= R \vdash M \textbf{ where } M \textbf{ is a molecule (multiset of } P\textbf{) and } R \textbf{ are the reactions (} D\textbf{)}
\end{aligned}
$$

- P is a process, which can either be a message sent on channel x, a definition of new names, or a parallel composition of processes.
- J is a join pattern that determines when reactions can be run. We can think of this as defining a new channel, and the reactions running as sending messages along channels (potentially more than one) at once.
- D consists of definitions in the form $J \triangleright P$ that match J to guarded process P. Definitions can be arbitrarily (nondeterministically) applied.

(d) Rules

$\rightleftharpoons$ represents structural congruence (as defined by the π-calulus) transformation rules (reversible and nondeterministic). Intuitively, these rules "dissolve" the solution enough that a reaction can take place, which is expressed in a single reduction rule ($\rightarrow$)

(Note: only the elements in both multisets participating in the rule are included)

$$
\begin{aligned}
\vdash P|Q &\rightleftharpoons \vdash P, Q \\
D \wedge E \vdash &\rightleftharpoons D, E \vdash \\
\vdash \textbf{def } D \textbf{ in } P &\rightleftharpoons D_\sigma \vdash P_\sigma \\
J \triangleright P \vdash J_\rho &\rightarrow J \triangleright P \vdash P_\rho
\end{aligned}
$$

σ substitutes fresh names for the defined channel names in D
ρ substitutes transmitted names for the formal (receiving) parameters in J
(instantiates the reaction rule)

(e) Examples

def $x\langle u \rangle \triangleright y\langle u \rangle$ **in** P

def $y\langle u \rangle \triangleright x\langle u \rangle$ **in def** $x\langle u \rangle \triangleright y\langle u \rangle$ **in** P **(requires renaming of innermost** x**)**

def $x_1\langle u \rangle | x_2\langle v \rangle \triangleright x\langle u, v \rangle$ **in** P **(multiplexing)**

def $once\langle\rangle | y\langle v \rangle \triangleright x\langle v \rangle$ **in** $y\langle 1 \rangle | y\langle 2 \rangle | y\langle 3 \rangle | once$ **(nondeterminism/**$once$ **is a lock)**

def $loop\langle\rangle \triangleright P | loop\langle\rangle$ **in** $loop\langle\rangle | Q$

def $newCCSchannel() \triangleright$
  def $send(v) | receive() \triangleright$ **return to** $send$**|return** v **to** $receive$
  in return $send, receive$
in $\cdots$

(f) Join Calculus

Process calculus produced from RCHAM: terms are the molecules of RCHAM, and structural equivalence/transition rules correspond to RCHAM calculus.

Structural Equivalence: P and Q are the same up to alpha-conversion and rearrangement of unguarded subterms that preserve bindings

$$\vdash P \rightleftharpoons^* \ \vdash Q \textbf{ means } P \equiv Q$$

Labelled Transition System:

$$\textbf{Transitions } \xrightarrow{\delta} \textbf{ s.t. } \delta \in D \cup \{\tau\}$$

$$\forall D = x\langle u\rangle | y\langle v\rangle \triangleright R,\ x\langle s\rangle | y\langle t\rangle \xrightarrow{D} R[s/u, t/v]$$

τ is a silent transition: $\xrightarrow{\tau}$ contains exactly the pairs of processes P, Q up to $\equiv$ s.t. $\vdash P \rightarrow \ \vdash Q$

If $P \xrightarrow{\delta} P'$**:**

$$\frac{}{P|Q \xrightarrow{\delta} P'|Q}$$

$$\frac{}{\textbf{def } D \textbf{ in } P \xrightarrow{\delta} \textbf{def } D \textbf{ in } P'} \quad (fv(D) \cap dv(\delta) = \emptyset)$$

$$\frac{}{\textbf{def } \delta \textbf{ in } P \xrightarrow{\tau} \textbf{def } \delta \textbf{ in } P'} \quad (\delta \neq \tau)$$

$$\frac{}{Q \xrightarrow{\delta} Q'} \quad (P \equiv Q \cap P' \equiv Q')$$

4 JoCaml: A Practical PL based upon the Join Calculus — Mandel & Maranget, 2007

JoCaml is the latest implementation of the join calculus. It is a distributed extension of OCaml. It takes from OCaml native-code and bytecode compilers, and extends OCaml, in the sense that OCaml programs and libraries are just a special kind of JoCaml programs and libraries.

spawn injects a molecule into the solution, and **def** ... **or** ... defines reactions.

(a) Mergesort example:

In mergesort, we want to break the array down into pairs, sort these pairs, and then merge the arrays. Note that we cannot merge the arrays in arbitrary orders: the "divide-and-conquer" approach will not work if we don't merge arrays of equivalent size.

We want a reaction to merge two arrays to form a molecule of the form **sorted[1;2;3;4]**. However, we cannot allow any two sorted molecules to merge. Instead, we have to form hierarchies based upon array length and somehow tell the CHAM to only merge particular molecules.

Note: the CHAM does not do any computations before starting a reaction, so reactions occur unconditionally.

Solutions:

i. Store information about which items have been merged (requires complicated data structure)

ii. Separate reactions for merging arrays of different sizes (merge smaller molecules to build new larger ones).

Problems: molecules have to be statically defined, so we can't know how many types of reactions to define until later!

Solution: recurse on locally defined molecules and reactions! Molecules cannot react with molecules produced in another recursive function call.

```
(* Attempt 1 *)
def mergesort(arr) =
    if (arr has length 1) then sorted(arr)
    else def sorted(x) & sorted(y) = sorted(array_merge x y)
    in let (arr1, arr2) = array_split arr
        in mergesort(arr1) & mergesort (arr2)
    (* results will be sorted(arr1') and sorted(arr2'), which will be merged recursively *)
```

This code will not work:

i. Cannot have a reaction with two identical molecules.

```
def a(x) & a(y) = c(x,y) => def a(x) & b() = a'(x) or a(x) & a'(y) = c(x,y)
```

ii. The name "sorted" will be undefined outside "mergesort," but we want to use mergesort recursively. If "sorted" is undefined outside "mergesort", we cannot use a reaction to combine the results of the reactions starting from "mergesort(arr1)" and "mergesort(arr2)".

The defined "sorted & sorted = sorted" reaction will never start because it is defined for the local "sorted" molecule instead of the (two different) "sorted" molecules that are locally defined inside the recursive calls to "mergesort(arr1)" and "mergesort(arr2)"

To fix this, it seems like we must define "sorted" outside of "mergesort". But if all "mergesort" reactions yield the same "sorted" molecule, the result of every "mergesort" will be free to combine arbitrarily with other results, and we lose the hierarchical organization of the merging reactions, which brings us back to our original problem.

To actually fix problem (2), we need to define two different molecules: one internal and one external of the mergesort.

- External: passed to "mergesort" molecule as a parameter (part of the constructor). "mergesort" then produces a molecule of external type.
- Internal: locally defined, passed to "mergesort" molecule as a parameter (part of the constructor), and used to sort the recursive results. These can only react with molecules defined within the same scope (i.e. at the same level of recursion).

```
def mergesort(arr, extern_sorted) =
  if Array.length arr <= 1 then extern_sorted(arr) else
  let (part1, part2) = array_split arr in
  (* define the reactions for the recursive calls *)
  def local_sorted(x) & a() = local_sorted'(x)
    or local_sorted(x) & local_sorted'(y) = extern_sorted(array_merge x y (<))
  (* call mergesort with the newly defined molecules, to be produced by the recursive calls *)
  in mergesort(part1, local_sorted) & mergesort(part2, local_sorted) & a()
  in (* mergesort is now defined; now we set up the initiating call *)
    def print_result (arr) = Printf.printf "finished:␣[%s]" (string_of_array arr string_of_int
    in (* now we call mergesort with an "extern_sorted" molecule argument of "print_result" *)
    spawn mergesort([| 3; 2; 5; 1; 4 |] , print_result)
```

(b) Observations:

i. Cannot detect the absence of a molecule in the solution.

ii. Cannot combine two solutions (running on two machines, for example) into one computation.

iii. All molecules have to be statically defined, so they cannot be computed dynamically or based on inputs. However, this reduces system errors from dynamically generating inconsistent reactions. e.g. injecting molecules that produce "out of bounds" errors is impossible.

iv. All "local_sorted" molecules are different molecules because of their different scope, and all reactions where a molecule appears must be in one scope. New reactions cannot be added to input molecules previously defined.

v. Reactions do not allow for some type of dynamically computed condition to dictate whether they run or not. However, the lack of inherent control flow can lead to overly complex solutions (no pun intended) when conditions need to be met before a reaction occurs.

vi. **def** creates locally defined molecules, but not locally defined reactions: all reactions are always available to the chemical machine.

vii. A function cannot evaluate to a molecule because molecules are not OCaml values.